Titre: Title:	Load balancing for distributed engineering applications
Auteur: Author:	Daojun Liu
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Liu, D. (2003). Load balancing for distributed engineering applications [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7134/

In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

UNIVERSITÉ DE MONTRÉAL

LOAD BALANCING FOR DISTRIBUTED ENGINEERING APPLICATIONS

DAOJUN LIU
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
SEPTEMBRE 2003

Your file Votre référence
ISBN: 0-612-86412-X
Our file Notre référence
ISBN: 0-612-86412-X

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

LOAD BALANCING FOR DISTRIBUTED ENGINEERING APPLICATIONS

présenté par: LIU DAOJUN

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BERTRAND François, Ph.D., président

M. GUIBAULT François, Ph.D., membre et directeur de recherche

M. ROY Robert, Ph.D., membre

ACKNOWLEDGMENTS

Nous tenons à remercier très sincèrement notre directeur, M. François Guibault pour le support, les idées et les conseils qu'il a su nous prodiguer tout au long de l'élaboration de ce travail.

Nous remercions très sincèrement M. François Bertrand et Robert Roy pour avoir accepté de siéger sur le jury du mémoire et pour leurs excellents conseils qui ont permi d'améliorer significativement le présent document.

Nous remercions très sincèrement M. Jean-Yves Trépanier pour le support, les idées et les conseils qu'il a su nous prodiguer tout au long de l'élaboration de ce travail.

Nous remercions également les professionnels de recherche et les étudiants du CERCA pour leur support.

Thanks very much to my director, François Guibault, who gave me a lot of supports, ideas and advices during my research work.

Thanks very much to Mr. François Bertrand and Robert Roy for participating in the jury and for their excellent advices.

Thanks very much to Mr. Jean-Yves Trépanier, who gave me a lot of supports, ideas and advices during my research work.

Thanks to the researchers and students who have given me supports during my work.

RÉSUMÉ

La conception de l'architecture d'une infrastructure logicielle pour le design en ingénierie doit être flexible, adaptable et robuste. La superposition des différents services offerts par l'infrastructure sous forme de couches logicielles en facilite l'évolution en fonction de besoins qui ne cessent de changer et de se raffiner. Le projet au sein duquel le présent travail a été mené a pour objectif l'automatisation des processus de design afin de permettre une meilleure collaboration entre les ingénieurs des différents départements d'une entreprise. Le fruit de cet effort a pris la forme d'un environnement d'exécution de tâches d'analyse nommé VADOR, qui a été développé et déployé chez Bombardier Aerospace.

Ce travail présente des besoins spécifiques pour la répartition de charge lors de l'exécution et de l'optimisation de tâches multiples sous forme de processus d'analyse en ingénierie effectués sur des machines hétérogènes dans un système distribué. Les algorithmes de répartition statique et dynamique des charges sont discutés et des données expérimentales acquises à l'aide d'une version opérationnelle du système y sont présentées et analysées. L'algorithme dit "Random" a été implanté afin de représenter les méthodes de répartition statique de charges. Cet algorithme de répartition a été comparé à quatre autres algorithmes de répartition dynamique de charge, qui intègrent les caractéristiques des applications d'ingénierie et de l'environnement d'exécution. Les algorithmes "Threshold" et trois versions de l'algorithme "Shortest" ("Shortest", "Shortest-Capacity", "Shortest-CPU") ont été implantés. Les méthodes "Shortest", "Shortest-Capacity" et "Shortest-CPU" visent à choisir la machine qui a la charge la plus faible. La différence principale entre les trois méthodes tient au fait qu'elles utilisent des indices de charge différents. L'algorithme "Shortest" utilise l'état de charge du CPU, de la mémoire et des

ressources d'entrées/sorties (E/S) comme indice de charge. L'algorithme "Shortest-Capacity" utilise l'état et la capacité du CPU, de la mémoire et des ressources E/S comme indice de charge. De son coté, l'algorithme "Shortest-CPU" utilise uniquement l'état de charge du CPU comme indice de charge. Alors que l'algorithme "Threshold" utilise l'indice de charge, comme pour la méthode "Shortest", il tient en plus compte de la valeur d'état "Threshold" pour sélectionner la machine la moins chargée. Tous les algorithmes de répartition dynamique de charges utilisent les caractéristiques de l'application dans le processus de décision.

Nous avons effectuées nos expériences dans plusieurs configuration de charge du système et nous avons analysé les résultats sous des charges de travail qui font collectivement intervenir un ensemble de tâches caractérisées par différents types d'utilisation des ressources (CPU-bound, memory-bound et I/O-bound).

À partir des résultats expérimentaux, nous avons constaté que les algorithmes dynamiques de répartition de charge améliorent tous la performance du système en comparaison avec l'algorithme "Random". Nous avons également constaté que l'utilisation de l'état de charge d'un plus grand nombre de ressources (CPU, mémoire et E/S) mène à une meilleure performance que celle où seulement une partie des capacités (ex. la charge du CPU seulement) est utilisée. Pour notre environnement expérimental, nous avons par ailleurs trouvé que l'algorithme qui utilise les capacités de la machine hôte n'améliore pas significativement la performance du système en comparaison des algorithmes qui utilisent uniquement l'état de charge de la machine. Nous avons aussi constaté que la méthode "Threshold" offre une performance similaire à celle de la méthode "Shortest".

ABSTRACT

The design of a framework architecture for engineering design must allow for flexibility, scalability, and robustness. Layering of the different services proposed by the framework allows for an easier evolution of the framework as the needs evolve. Our project aims to automate the design process between engineers among different departments.

This work indicates that in a distributed system, load balancing is a very important part of an optimizing multi-tasks execution environment. Static and dynamic load balancing algorithms are discussed and experimental data has been acquired through the use of a production version of the VADOR framework. The Random algorithm was implemented as the representative of static load balancing methods, and the application and system characteristics are integrated into four distinct dynamic load balancing algorithms. As for dynamic load balancing algorithms, the Threshold and three versions of the Shortest (Shortest, Shortest-Capacity, Shortest-CPU) algorithms have been implemented. The Shortest, Shortest-Capacity and Shortest-CPU methods try to select a host with the lightest load value. The difference between the three methods is that they use different load indexes. The Shortest algorithm uses the CPU, memory and I/O resource load status as load index. The Shortest-Capacity algorithm uses the CPU, memory and I/O resource status and their capacities as load index. The Shortest-CPU algorithm only uses CPU load state as the load index. The Threshold algorithm use load index as in the Shortest algorithm, but also uses a Threshold state value to select a host. All dynamic load balancing algorithms use the application characteristics in the decision-making.

Experiments have been performed in different system load conditions and results

have been analyzed under workloads, which mix CPU-bound, memory-bound and I/O-bound tasks together. From the experiment results, we conclude that the load balancing algorithms which use the host load status improve the system performance when compared to the Random algorithm. And the utilization of more resources (CPU, memory and I/O) in the load status yields a better performance compared to using only part of the resources load status (eg. only CPU load). Under our experimental environment, we have found that the algorithm which uses the host capacities does not significantly improve overall system performance when compared to algorithms which only use the host load status. We have also found that the Threshold method has performance characteristics similar to those of the Shortest method.

CONDENSÉ EN FRANÇAIS

Introduction

La conception basée sur des méthodes numériques d'analyse est un domaine qui a évolué très rapidement au cours des vingt dernières années. Des technologies matures pour la simulation de systèmes physiques complexes sont maintenant disponibles dans la grande majorité des disciplines du génie. Dans ce contexte, le projet VADOR vise à développer une infrastructure logicielle permettant d'intégrer des applications d'analyse en ingénierie provenant de différentes disciplines. Le système VADOR cherche ainsi à fournir aux ingénieurs concepteurs un environnement multidisciplinaire d'exécution des applications d'analyse intégrant des modèles à différents niveaux de fidélité et capable de gérer les résultats des analyses produits au cours de la phase de conception préliminaire de grands systèmes, particulièrement en aéronautique.

VADOR vise à intégrer non seulement des applications commerciales, mais également un grand nombre d'applications patrimoniales, afin de construire des processus complexes d'analyse utilisant différentes applications dans un environnement de calcul distribué et hétérogène. L'objectif principal du présent mémoire est d'étudier différentes approches de répartition automatique de tâches qui soient adaptées aux environnements de calcul distribués et hétérogènes. Spécifiquement, le présent travail se propose d'examiner des approches de répartition de tâches qui tiennent à la fois compte des aspects dynamiques de charge des noeuds formant l'environnement d'exécution, des caractéristiques matérielles de chaque noeud, et des besoins en ressources des différentes applications. Les approches les plus prometteuses seront par ailleurs intégrées au système logiciel VADOR afin de fournir une méthode

transparente d'allocation des ressources de calcul lors de l'exécution des processus d'analyse.

Revue de la recherche pertinente

Puisque, au sein d'un système distribué, les tâches de calcul peuvent être générées à partir de n'importe quel noeud, une utilisation complète des capacités de calcul d'un système implique nécessairement un mécanisme d'affectation des tâches qui tente de répartir uniformément celles-ci au travers du réseau. Les algorithmes de répartition de charge (*load balancing*) tentent de solutionner le problème d'affecter les tâches provenant de différentes sources de façon uniforme aux différents serveurs dans un réseau, de façon à éviter des situations où certains serveurs seraient très peu chargés alors que d'autres seraient surchargés. Certains auteurs (Wang and Morris, 1985, Ferrari and Zhou, 1987) ont proposé différentes architectures pour la réalisation de systèmes de répartition de charge dans des systèmes distribués. Pour ce type de système, les tâches peuvent être réparties de façon centralisée ou de façon distribuée (Shiraze and Hurson, 1995) (Chapitre 5). Une approche pour classifier les algorithmes de répartition de charge distingue les algorithmes statiques des algorithmes adaptatifs, "Les algorithmes statiques de répartition de charge ne tiennent pas compte de l'état courant du système au moment de prendre une décision. Les algorithmes adaptatifs de répartition de charge réagissent aux changements dans l'état du système" (Shiraze and Hurson, 1995) (Chapitre 5).

Les algorithmes statiques typiques incluent les algorithmes *Random* (Wang and Morris, 1985,Cao *et al.*, 2000), *Round-Robin* (Wang and Morris, 1985), et les méthodes d'assignation basées sur la taille (*Size-based assignment* Harchol-Balter *et al.*, 1999, Crovella and Harchol-Balter, 1998, Harchol-Balter, 1999). Dans l'algorithme

Random, une tâche entrante est envoyée au noeud i avec une probabilité $1/h$, où h est le nombre de noeuds. Dans l'algorithme *Round-Robin*, les tâches sont assignées aux noeuds de façon cyclique. Les politiques d'assignation basées sur la taille répartissent les tâches en se basant sur la distribution de la tailles des différentes tâches à traiter.

Une politique d'assignation basée sur le choix du meilleur noeud est utilisée par plusieurs algorithmes dynamiques de répartition de charge (Eager *et al.*, 1986, Svensson, 1990). L'algorithme *Threshold* définit des valeurs de garde pour le statut de chargement des noeuds candidats. La valeur de garde est utilisée pour déterminer si un noeud n'est chargé que légèrement, ou s'il est surchargé (Eager *et al.*, 1986). L'algorithme *Biding* divise le rôle décisionnel entre le répartiteur et les noeuds serveurs (Rommel, 1991, Musliner and Boddy, 1996).

Quelle information sera utilisée et quelle information est la plus pertinente nous amène à considérer différents types d'information de charge. Ferrari and Zhou, 1987 décrivent certaines informations de charge liées au processeur, soit la longueur de la file d'attente du processeur et l'utilisation moyenne du processeur au cours d'une récente période de temps. Zhou, 1987 et Kunz, 1991 proposent d'utiliser la longueur des files d'attente du processeur et du système d'entrées/sorties. Xiao *et al.*, 2000 mentionne une mesure impliquant une combinaison de données sur le processeur et la mémoire. Des combinaison impliquant des données sur le processeur, la mémoire et les entrées/sorties sont discutées dans Zhou, 1987, Devarakonda and Iyer, 1989, Ferrari and Zhou, 1987. Les avantages de solutions combinant des données sur plusieurs ressources paraissent évidents dans un contexte d'applications multidisciplinaires.

Dans un système distribué hétérogène, l'index de charge d'un noeud devrait inclure les caractéristiques spécifiques du noeud. Dans son article, Ezzat, 1986 inclut

l'utilisation moyenne du processeur et la capacité spécifique du processeur d'un noeud dans sa définition d'un index de charge. La puissance d'un processeur et la capacité en mémoire des différentes machines d'un réseau sont discutés par Xiao *et al.*, 2000, qui cherchent à quantifier l'hétérogénéité d'un système et à utiliser cette information comme index de charge. Pour les mêmes raisons, un effort de classification des noeuds a été entrepris dans la présente recherche, afin de quantifier la puissance des noeuds d'un réseau hétérogène, et ce pour chaque type de ressource mise à la disposition des tâches.

Il existe par ailleurs trois méthodes afin d'obtenir l'information de charge d'un système: à la demande (*demand-driven*), de façon périodique (*periodic collection*), ou en se basant sur les changements d'état (*state change driven*) (Lu and Liu, 1996).

En vue de tirer le maximum de performance de son système, un usager doit non seulement tenir compte des caractéristiques dynamiques de charge du système, mais également des besoins en ressources de ses applications, afin de développer une méthode de répartition qui satisfasse ses critères de performance (Berman *et al.*, 1996). Les applications ayant servi à construire l'ensemble d'applications test pour la présente recherche ont ainsi été classées en fonction du type de ressources principalement utilisées par l'application, et certains des algorithmes testés ont pris en compte cette classification.

Répartition de charge pour le système VADOR

Le système VADOR

Le système VADOR vise à intégrer différents types d'applications d'ingénierie provenant d'un grand nombre de départements techniques, ou de disciplines. Le système VADOR inclut une interface usager graphique (*GUI*), un serveur d'exécution (*Executive Server*), des serveurs encapsulant les applications (*Wrapper Servers*), et un serveur gestionnaire des données (*Librarian Server*). Au travers de l'interface graphique, les usagers peuvent créer des tâches, contrôler l'exécution de tâches, et surveiller le statut du système. Le serveur de gestion des données permet de découpler les interfaces de programmation du système VADOR du système de base de données sous-jacent, et des fonctionnalités associées aux serveurs de fichiers. Le serveur d'exécution de VADOR contrôle l'exécution des différentes tâches, tandis que les serveurs encapsulant les applications servent à réellement exécuter les applications d'analyse. Les tâches sont ainsi transférées du serveur de contrôle de l'exécution aux serveurs encapsulant les applications afin d'être exécutées, et les résultats sont retournés au serveur de contrôle.

Répartition de charge dans VADOR

Lorsque le serveur d'exécution reçoit une tâche, il doit déterminer, parmi l'ensemble des serveurs où l'application est disponible, celui auquel la tâche sera effectivement envoyée. La charge sur les serveurs d'application varie dynamiquement, l'algorithme de répartition de charge doit donc tenter de répartir le plus uniformément possible la charge sur le réseau, en tenant compte dynamiquement de la charge et des caractéristiques de l'application.

Afin de mieux orienter les algorithmes dynamiques, les tâches sont classées soit comme limitées par le processeur (*CPU-bound*), limitées par la mémoire (*Memory-bound*) ou limitées par les entrées/sorties (*I/O-bound*).

Algorithme de répartition de charge statique

L'algorithme *Random* a été implanté afin de représenter les algorithmes de répartition de type statique. Les caractéristiques de la tâche et l'information de charge du système ne sont pas considérés. Lorsqu'une tâche arrive, un serveur est choisi aléatoirement parmi les serveurs où l'application est disponible.

Algorithme de répartition de charge dynamique basé sur la charge

Dans cette étude, quatre algorithmes de répartition de charge dynamique sont étudiés.

L'algorithme de répartition de charge basé sur la charge dynamique la plus faible (algorithme *Shortest Dynamic* ou simplement *Shortest*), n'utilise que la charge courante de chaque noeud afin de produire une assignation des tâches. Le pourcentage de temps processeur inutilisé est choisi comme indicateur de charge du processeur. La quantité de mémoire libre est utilisée comme charge de la mémoire. Le nombre de lectures et d'écritures entre les tampons en mémoire et le disque est choisi comme indicateur de charge du système de disque.

Sommairement, l'algorithme est le suivant:

1. récupérer les paramètres de la tâche en terme des limites sur les ressources au niveau de la mémoire;

2. choisir les noeuds qui satisfont aux limites en mémoire de la tâche;
3. Si l'exécution de la tâche est limitée par son utilisation du processeur, choisir la charge processeur comme indicateur de charge;
4. Si l'exécution de la tâche est limitée par son utilisation de la mémoire, choisir la charge mémoire comme indicateur de charge;
5. Si l'exécution de la tâche est limitée par son utilisation des entrées/sorties, choisir la charge des entrées/sorties comme indicateur de charge;
6. Calculer la charge de l'ensemble des noeuds choisis selon l'indicateur sélectionné;
7. Choisir le noeud le moins chargé, c'est-à-dire dont l'index de charge est le plus petit.

Dans l'algorithme de répartition de charge basé sur la charge dynamique et la capacité des noeuds (*Load-capacity*), l'algorithme tient en plus compte de la puissance des noeuds dans son assignation. Les indicateurs de charge sont redéfinis en multipliant les indicateurs de l'algorithme *Shortest* par la capacité du noeud. Cet algorithme utilise ensuite le même algorithme de sélection de noeud que l'algorithme *Shortest*.

L'algorithme de répartition de charge basé sur la charge dynamique du processeur (*Load-CPU*) n'utilise que la charge du processeur pour exprimer l'indicateur de charge.

L'algorithme de répartition de charge basé sur la charge dynamique avec valeur de garde (*Threshold*) utilise les indicateurs de charge décrit pour l'algorithme *Shortest*, mais sélectionne un noeud uniquement si l'indicateur de charge est supérieur à la valeur de garde.

Stratégie d'échange d'information

Afin d'accumuler l'information sur leur propre charge, chaque noeud procède à des séries de collectes périodiques sur des intervalles de temps fixe. Une période de collecte d'une durée P est subdivisée en n segments. Sur chaque segment $t = P/n$, les indicateurs de charge du processeur, de la mémoire et des entrées/sorties sont calculées, et les n valeurs des indicateurs sont accumulés. Les indicateurs retournés sont les moyennes des n indicateurs sur la période P.

Implantation du système

Tel qu'illustré à la figure 1, le sous-système de répartition de charge comprend deux parties: le collecteur global et les collecteurs locaux.

Le collecteur global est imbriqué au niveau du serveur central (*Executive Server*). Son rôle consiste à rassembler la charge des noeuds du réseau et à assigner les tâches aux différents noeuds selon un des algorithmes de répartition mentionnés précédemment.

Le collecteur global comprend trois modules. Le module de communication est responsable de gérer les échanges de données entre le collecteur global et les collecteurs locaux sur les différents noeuds du système. Le module de gestion des indicateurs de charge reçoit les informations de charge des différents noeuds du réseau, et les stocke dans ses structures de données internes. Le module d'assignation de tâche sert à choisir un noeud du réseau pour l'exécution d'une tâche.

Un collecteur local réside sur chaque noeud afin d'accumuler l'information de charge sur le noeud et de transmettre périodiquement cette information au collecteur global. À la fin de chaque période de collecte, une estimation est faite afin

de déterminer si un changement significatif de charge a été observé par rapport aux dernières informations transmises, et si c'est le cas, les nouvelles informations sont envoyées au collecteur global; dans le cas contraire, les nouvelles informations de charge sont oubliées, et la collecte reprend.

Le collecteur local est composé de trois modules: le module de communication, le module de traitement des messages et le module de collecte de charge. Le module de communication échange les informations avec le collecteur global. Le module de traitement des messages reçoit et manipule les messages en provenance du collecteur global. Enfin, le module de collecte est responsable de l'accumulation des informations de charge du noeud.

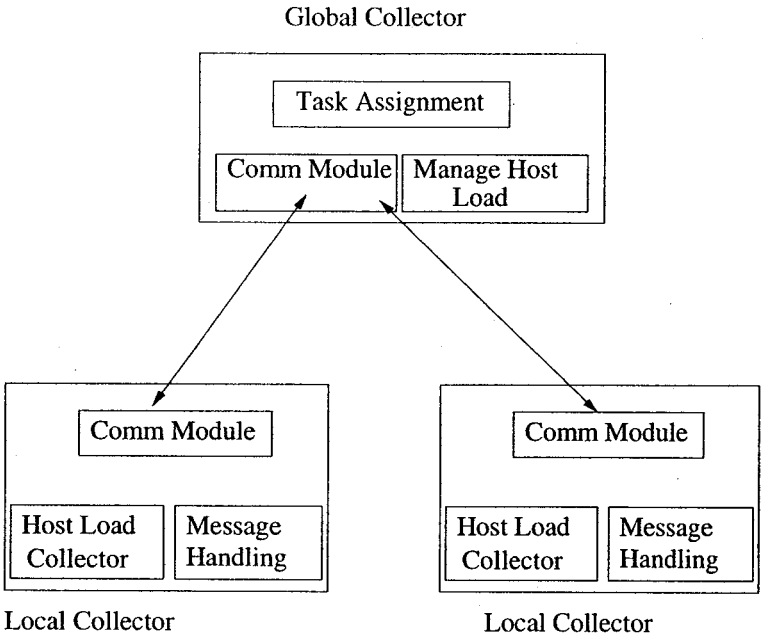


Figure 1 Architecture du système de répartition de charge

Comparaison des algorithmes de répartition de charge dans VADOR

Afin de comparer la performance des différents algorithmes dans le contexte du projet VADOR, des tests ont été effectués sur deux types de systèmes. Le premier est un système composé de noeuds homogènes connectés à un réseau haute vitesse, alors que le second système est composé de noeuds hétérogènes connectés sur un réseau standard.

Pour l'ensemble des tests effectués, les programmes choisis effectuaient tous une quantité significative de calcul, typique des applications scientifiques pour lesquelles le système VADOR a été conçu. Par ailleurs, les coûts de communication des applications étaient tous très petits par rapport aux coûts de calcul, et, durant leur exécution, aucun des programmes choisis n'avait à communiquer avec d'autres tâches. Considérant ceci, il a été décidé de négliger l'influence du trafic sur le réseau.

Les métriques choisies pour mesurer la performance des systèmes sont le temps de réponse moyen et le ralentissement moyen. Le temps de réponse est le temps nécessaire à une tâche arrivant sur un noeud pour terminer son exécution. Le ralentissement est le rapport entre le temps de calcul utilisé et le temps total utilisé pour l'exécution d'une tâche. Le temps de réponse moyen est défini comme la moyenne des temps réponse de toutes les tâches et le ralentissement moyen est la moyenne du ralentissement de toutes les tâches.

Plusieurs cédules ont été définies pour simuler différents chargements des systèmes. Les cédules ont été classées en légères, moyennes et lourdes. La commande *sleep* a été utilisée pour simuler les intervalles de temps entre les arrivées de tâches.

Les programmes suivants ont été utilisés pour construire la charge de travail:

Queens: Le programme *Queens* calcul le temps pris par un ordinateur pour trouver toutes les façons de placer n reines sur un échiquier de nXn de manière à ce qu'aucune reine ne puisse en prendre une autre. L'exécution de ce programme est limitée par son utilisation du processeur (Netlib, 2002);

Pi: Ce programme calcul le nombre *pi* à une précision arbitraire. On contrôle le temps d'exécution du programme en configurant le nombre de décimales désirées du nombre *pi*. L'exécution de ce programme est limitée par son utilisation du processeur;

Optim-NURBS (O-N): Ce programme optimise une courbe NURBS représentant un profile d'aile d'avion. L'exécution de ce programme est limitée par son utilisation du processeur;

Matrix multiplication (m-m): Ce programme est une application standard effectuant la multiplication de matrices. Ce programme est limité par son utilisation de la mémoire (Xiao *et al.*, 2002);

Hanoi: Le programme *Hanoi* est un programme en nombres entiers qui résout le problème des tours de Hanoi. Ce programme a été modifié pour qu'il produise de grandes quantités de résultats sous la forme d'un fichier et qu'il soit ainsi limité par les entrées/sorties;

SumTwist: est un petit programme qui calcule des courbes. Il a été configuré de façon à être limité par son utilisation du processeur;

Loopsh: Il s'agit d'un programme construit spécifiquement pour les besoins des tests qui produit en sortie n fois un message sous la forme d'une chaîne de caractères. Ce programme est limité par ses entrées/sorties.

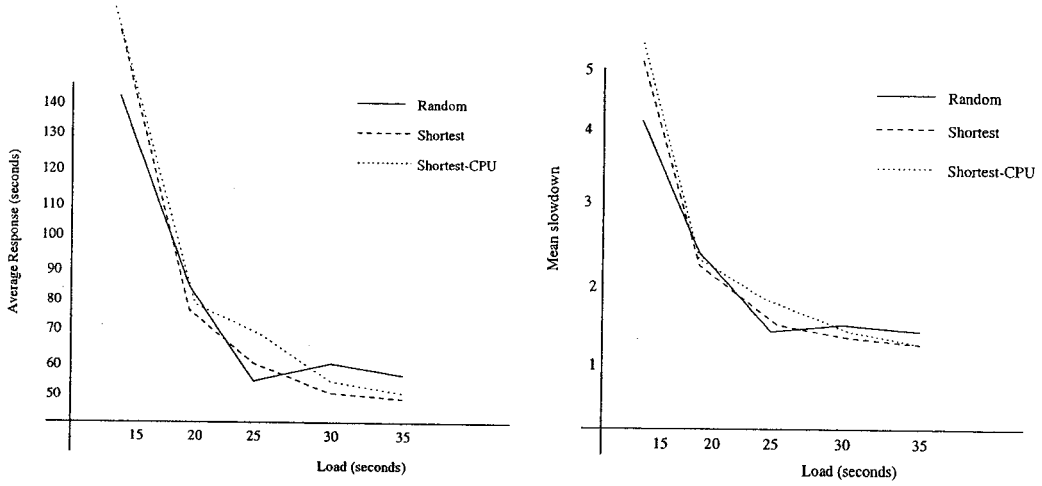
Tests sur le système homogène

La première série de tests de répartition de charge a été effectuée sur un système homogène. Ce système comprenait quatre stations rapides, chacune équipée d'un seul processeur de la famille x86 connectées à un réseau de communication rapide. Chaque station était équipée de 1Gb de mémoire vive, et le système entier partageait un même système de fichiers. Le système d'exploitation utilisé était Linux 2.4.7.

Les programmes Queens, Pi, Matrix multiplication, Hanoi et Loopsh ont été utilisés pour produire les charges de travail.

La figure 2 présente les résultats pour trois des algorithmes de répartition de charge, soit *Random*, *Shortest* et *Shortest-CPU* pour cet environnement de test. Ces trois algorithmes se comportent de façon pratiquement identique. Lorsque le système est extrêmement chargé, l'algorithme *Random* se comporte légèrement mieux que les autres. Par contre, lorsque le système n'est pas surchargé, l'algorithme *Shortest* se comporte légèrement mieux que l'algorithme *Shortest-CPU*, qui lui-même se comporte mieux que l'algorithme *Random*. Bien que les algorithmes de répartition de charge puissent améliorer légèrement les performances sur un système homogène très rapide, les améliorations restent marginales, et les différences entre les algorithmes ne sont pas significatives.

Lorsque la fréquence d'arrivée des tâches dans le système devient très grande, l'information sur la charge stockée au niveau du collecteur global se périme rapidement. Lorsque ces informations périmées sont ensuite utilisées pour évaluer les noeuds, des assignations incorrectes sont effectuées. Dans ces circonstances, il est facile de considérer un noeud comme relativement peu chargé et de lui envoyer des tâches alors qu'il est en fait déjà surchargé.



(a) Le temps de réponse moyen.

(b) Le ralentissement moyen.

Figure 2 Résultats pour le système homogène.

Tests sur le système hétérogène

Des expériences ont été effectuées sur un système hétérogène distribué comprenant six noeuds, mais ces expériences pourraient facilement être étendues à un système comprenant N noeuds. Les six noeuds sont des noeuds connectés à un réseau local et gérer par un serveur central. Des serveurs encapsulant les applications ont été installés sur chacun des noeuds. Les six noeuds étaient des machines SGI équipées de processeurs MIPS dont la vitesse et les caractéristiques en mémoire et au niveau du système de disque variaient. Tous les noeuds utilisaient le système d'exploitation IRIX6.5.

Les valeurs obtenues pour chaque machine sur le banc d'essai standard SPECint95 ont été utilisées pour exprimer la puissance des noeuds (DiMarco, 1997). La taille totale de la mémoire a été utilisée pour exprimer la capacité mémoire. Pour la puissance du sous-système d'entrées/sorties, la bande passante de chaque machine a été évaluée à l'aide d'expériences utilisant la commande *cp* pour copier des fichiers de différentes tailles et mesurer le temps d'exécution pour chaque noeud.

Pour l'algorithme *Threshold*, des valeurs de garde sont nécessaires pour configurer l'algorithme et distinguer les différents états d'un noeud (légèrement chargé ou surchargé). La valeur de garde utilisée pour la puissance du processeur a été fixée à 20% du temps du processeur non utilisé. La valeur de garde pour la mémoire a été choisie à 50MB selon la méthode décrite par Xiao *et al.*, 2002. Pour le sous-système d'entrées/sorties, des expériences ont été menées pour différentes configuration de charge et différentes valeurs de garde. Suite à ces expérience, une valeur de garde de 700 pour le nombre d'entrées/sorties a été choisi.

Les figures 3 (a) et (b) présentent les performances de cinq algorithmes soit *Random*, *Shortest*, *Shortest-Capacity*, *Shortest-CPU* et *Threshold* pour plusieurs charges du système.

À l'examen de la figure 3 (a), et en considérant le temps réponse moyen d'un groupe de tâches, on constate que tous les algorithmes de répartition de charge dynamique offrent de meilleures performances que l'algorithme *Random*. La principale raison pour laquelle l'algorithme *Random* offre de si piètres performances réside essentiellement dans le fait que cet algorithme n'a aucune connaissance de l'état du système (Wang and Morris, 1985). L'algorithme *Random* présente cependant une performance similaire à celle des algorithmes *Threshold* et *Shortest-CPU* lorsque le système devient extrêmement chargé. Lorsque tous les noeuds sont surchargés, aucune tâche arrivant sur le système ne peut être traitée normalement, et les tâches s'accumulent dans le système, le temps réponse augmente alors de façon significative. La performance obtenue des algorithmes *Shortest* et *Threshold* ne diffèrent pas significativement. La méthode *Shortest* assigne les tâche au noeud qui est le moins chargé. Par contre, l'algorithme *Threshold* vérifie parmi un groupe de noeuds afin de trouver un noeud qui ne soit pas surchargé (Eager *et al.*, 1986). Comme le montre la figure 3 (a), il n'y a pas de différence significative entre les

algorithmes *Shortest*, *Threshold* et *Shortest-Capacity*.

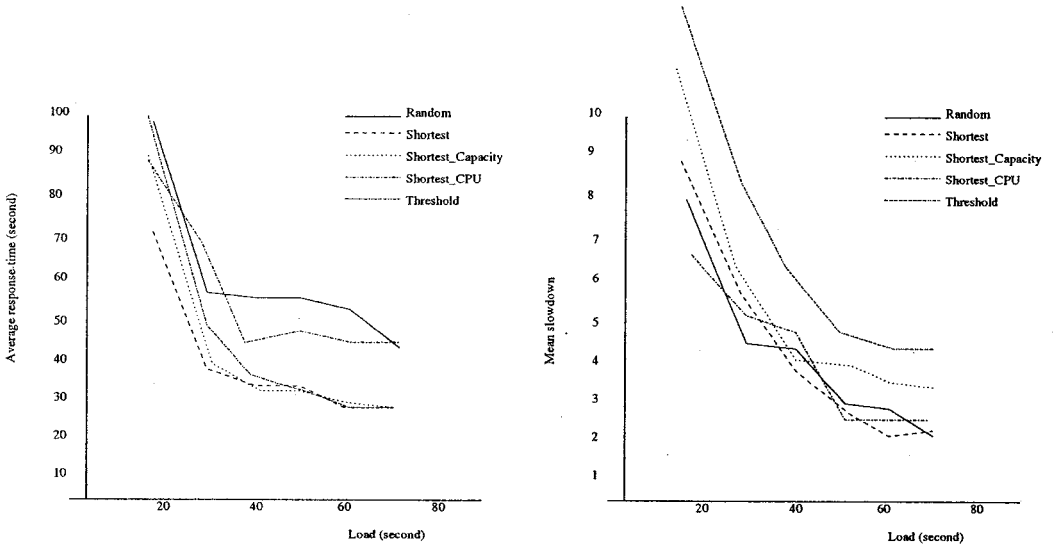
Une comparaison des algorithmes *Shortest*, *Threshold* et *Shortest-Capacity* avec l'algorithme *Shortest-CPU* tend à montrer que ce dernier algorithme offre de moins bonnes performances que les trois premiers. L'algorithme *Shortest-CPU* n'est pas en mesure d'assigner correctement des tâches limitées soit par leur utilisation de la mémoire ou des entrées/sorties. Bien qu'un noeud ne soit que légèrement chargé au niveau du processeur ne signifie pas nécessairement qu'il le soit également au niveau de la mémoire ou des entrées/sorties.

La figure 3 (b) illustre le ralentissement moyen pour un groupe de tâches pour différents algorithmes. Les algorithmes *Shortest-CPU*, *Random* et *Shortest* présentent des meilleures performances que les algorithmes *Threshold* et *Shortest-Capacity*. Au sein de l'environnement de test, l'un des noeuds était beaucoup plus puissant que les autres. Ce noeud puissant était en mesure de traiter les tâches beaucoup plus rapidement que les autres noeuds. Lorsque plus de tâches étaient assignées au noeud puissant, le ralentissement moyen des tâches augmentait. Cependant, la puissance de ce noeud permettait d'exécuter les tâches très rapidement.

Comparaison entre les systèmes homogène et hétérogène

Les résultats obtenus sur le système homogène diffèrent très significativement de ceux obtenus sur le système hétérogène. La performance des cinq algorithmes testés différent significativement dans le cas du système hétérogène, alors que ces mêmes algorithmes se comportent tous de façon pratiquement identique sur le système homogène.

Sur un système homogène où tous les noeuds sont puissants, les tâches ne sont pas retardées longtemps sur les noeuds. Chaque noeud a une capacité impor-



(a) Le temps de réponse moyen.

(b) Le ralentissement moyen.

Figure 3 Résultats pour le système hétérogène.

tante de traitement et les tâches peuvent être traitées immédiatement, ce qui évite d'atteindre un état où la charge serait débalancée entre les noeuds. Des algorithmes sophistiqués de répartition de charge deviennent alors moins nécessaires, dans la mesure où tous les noeuds sont dans des états semblables ou identiques, et pour lesquels un algorithme simple comme *Random* se comporte correctement.

Développements futurs

Bien que fonctionnel, le système de répartition de charge n'est présentement pas en mesure de mettre en queue les tâches lorsque le système devient surchargé. Parmi les cinq algorithmes étudiés, seul l'algorithme *Threshold* permettrait de mettre en queue les tâches. En testant le système en entier, l'algorithme *Threshold* pourrait déterminer que le système est surchargé et mettre en queue les tâches supplémentaires. Lorsque la charge diminuerait sur l'un des noeuds, l'algorithme

pourrait en être averti et poursuivre l'assignation des tâches. Une telle approche permettrait d'éviter de congestionner le système en entier en cas de surcharge, et de façon globale, permettrait d'améliorer les performances du système.

Plusieurs paramètres interviennent dans la configuration des collecteurs locaux, qui doivent périodiquement récolter de l'information de charge sur les noeuds et la transmettre au collecteur global en cas de changement significatif. Une étude plus approfondie devrait être menée afin de déterminer les valeurs optimales de ces paramètres en fonction des caractéristiques matérielles du système.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF NOTATIONS AND SYMBOLS

CERCA	CEntre de Calcul en Recherche Appliqué
VADOR	Virtual Aircraft Design Optimization framewoRk
CFD	Computational Fluid Dynamics
CSM	Computational Structural Mechanics
MDO	Multidisciplinary Design Optimization
NURBS	Nonuniform Rational B-Splines

INTRODUCTION

Computation based design is a rapidly evolving field. Mature technologies for the simulation and analysis of complex physical systems are now available in most engineering disciplines. In aeronautical applications, the aerodynamics and structures disciplines each propose their own state-of-the-art methods for respectively simulating the behavior of fluid flow and of the stress and strain of structures: high-fidelity computer codes now solve the 3D Navier-Stokes equations for a full aircraft in computational fluid dynamics (CFD) and construct complete finite-element models for a full aircraft in computational structural mechanics(CSM). In parallel to the development of these state-of-the-art high fidelity models for simulating physical systems, design methods have been developed to help design engineers in making the best possible design within some constraints. As the high fidelity models in isolated disciplines have matured, the development of Multidisciplinary Design Optimization(MDO) has risen to develop methodologies and tools to tackle the formidable challenges of integrating high-fidelity physical models in a design environment and allow the synergism of mutually interacting disciplines to be fully exploited. The challenge in applying an MDO methodology in a large aeronautical corporation mainly lies in organizational aspects of engineering design. Within these companies, and mostly for historical reasons, design departments are often strongly segregated by disciplines with each department being responsible for specific aspects of the engineering work required, for example, to design an airplane. As a result, challenges arise when different departments can not effectively communicate and share information among them. In order to tackle these challenges, a current trend consists in providing the engineers with integrated software environments supporting collaboration and enabling MDO (Alzubbi *et al.*, 2000).

Under the sponsorship of Bombardier Aerospace, the VADOR project aims to develop a framework for integrating multi-disciplinary and multi-fidelity engineering analysis programs and for managing the analysis results. Starting from general considerations on computation based design and the MDO methodologies, the VADOR framework proposes a well structured methodology to integrate various distributed applications together and promote cooperation among engineers from different departments.

The design of a framework architecture for engineering analysis and design must allow for flexibility, scalability, and robustness. The VADOR system architecture is composed of : The VADOR GUI through which most users will interact with the system; the VADOR Librarian which is a server that allows to decouple the API from the underlying database server and file servers capabilities ; the VADOR Executive server which manages the execution of various tasks; the VADOR wrapper servers which run wrapped legacy analysis packages.

In the VADOR system, the multi-disciplinary and multi-fidelity engineering analyses programs are managed by the system framework. The engineers in different departments define and create their own tasks through the GUI. When the tasks arrive into the system, they are automatically sent to the Executive server where it is decided how to run and control them. The Executive server receives the incoming task and makes a decision to assign it to a proper Wrapper server. On the Wrapper server, the tasks are executed as different program packages. The VADOR system connects various departments in Bombardier Aerospace, and automates tasks across them.

The VADOR Executive Server receives multiple tasks and must choose a Wrapper server to run them. In a distributed application system with heavy burden, load unbalance will arise during the running of the system. Some hosts may be in a

heavy load with very poor performance, while others run under a light load or are in an idle state. In order to use the entire system's resources effectively, a load balancing algorithm should be incorporated into the system. It schedules tasks into Wrapper servers automatically by considering their state information, and aims to evenly separate burden among Wrapper servers.

In a general sense, users of a distributed system expect to effectively share the information and the resources in that system. More importantly, a distributed system promises to improve the performance of the overall computing infrastructure, as users gain access to the computational power of more than a single processor. In such a context, load balancing mechanisms must be used to spread the workload evenly across the distributed system (Cao *et al.*, 2000).

In this project, we aim to develop load balancing methods that properly take advantage of system load information. However, load information is spread across the entire system, how and when to collect this information thus becomes an important issue that should be carefully considered. While some general discussions on this theme can be found, for instance in Shiraze and Hurson, 1995, developing a suitable model that accounts for the specific properties of the VADOR environment is our main objective. A load balancing algorithm is a mean to determine where an arriving task is to be executed. In terms of the system architecture of our project, a central host is designated as the scheduler which receives updated load information from all other hosts and assembles that information into a load vector. When a task is transferred, the central host calculates the relative host loads and then selects the best one to assign the task. How to send the load information to the central host is one aspect of the load balancing problem that we should resolve. Constructing an algorithm based on this information is the second aspect. Since we are dealing with a heterogeneous system, to be effective, collection and load distri-

bution will need to account for performance differences among nodes. Calibration of nodes is thus an important task that must be dealt with before hand. At the same time, we should integrate this subsystem seamlessly with other parts of the VADOR project. To this end, we will utilize existing communication mechanisms between the Executive Server and Wrapper Servers.

In a distributed system, the potential for resource sharing and its possible rewards are substantial. In order to share these resources effectively, some measures of the loads being imposed on the resources has to be made available to the clients. The information about the resource load is part of the system's state, and changes rapidly (Shiraze and Hurson, 1995). The load information is a quantitative measure of a host's load. It serves as one of the most fundamental elements in the load balancing process. The efficiency of a load balancing scheme can be affected heavily by how the load information is chosen and when the information is updated. An empirical evaluation of several load information indexes was listed out. Generally, the load information should take into account not only the CPU needs of a task, but also its I/O and memory operation requirements (Shiraze and Hurson, 1995). In our project we shall collect these informations. We will investigate various methods of using this information and determine the most appropriate algorithm which will make use of this information to dispatch tasks. Experiment tests will be carried out to demonstrate the availability of the load information. At the same time we also work on how to use all the information together, and decide the weight of each load information according to the specifics of our project. Some means of correlating load information with node capacity will be studied. For example, the CPU may be heavily congested, while the disks are not. In this case, to an incoming CPU-bound job the host's load is very high, whereas to an incoming I/O-bound job the host's load is low, because it will not experience much queuing at the disks. In our project, several Unix system services are considered to collect

host load information, and these informations will be balanced with the coming task's characteristics to improve host selection.

"Resource-usage prediction can be a sound basis for load balancing in a distributed computer system" (Devarakonda and Iyer, 1989). This project develops a statistical approach for predicting the resources a task will use. We collect the resource-cost information with each task under different computing environment and then summarize them to make an estimation for the task. While tasks select hosts to execute, these informations are combined into host's load. These informations are used to identify the task and give a simple prediction to users. It is helpful for the user to understand and control the task execution. In our model, the CPU-time, memory occupied, and disk I/O information of each task are collected and analyzed. Tasks are classified into different groups which represent different classes of system resources usage. When a task comes, this information is combined into the decision making algorithm.

For a distributed system, many load balancing algorithms have been brought out. In our project we will implement five algorithms (Random, Dynamic shortest resource utilization without system capacities, Dynamic threshold resource utilization without system capacities, Dynamic shortest resource utilization with system capacities, and Dynamic shortest with CPU utilization) and compare their performances in two different computing environments. One environment is a homogeneous high speed computing network. The other is a heterogeneous computing network which includes workstations with different CPU, memory and I/O capacities. Based on these experimental results we will analyze the characteristics of the various algorithms and choose the best one for our project.

In the following chapter we review related research on load balancing. Chapter 2 describes how to measure machine capacities. Chapter 3 gives an overview of our

system model, load balancing algorithms and load information collection. Some tests are detailed out in Chapter 4. In the final part we present our conclusions.

CHAPTER 1

REVIEW OF RELATED RESEARCH

1.1 System architecture

The advantages of a distributed computer system over a single powerful general-purpose computing facility are multiple, these include higher availability, better extensibility, and increased overall performance. In order to fully take advantage of these characteristics, efficient algorithms for the system-wide control of resource sharing are needed. Further more, some disadvantages still exist in the distributed system, such as, security problem, not easy to control when a workstation fails down.

In a distributed system, tasks are created in hosts across the network. In order to utilize the entire system capability, the tasks should be dispatched in the network evenly. Wang and Morris, 1985 presented a model of a load balancing system. In that model a logical review of load balancing systems was given out. Tasks enter the system through the hosts called *sources* and are scheduled to hosts called *servers*. A single host might be both a source and a server. Load balancing algorithms address the problem of how to schedule the tasks coming from various sources into the servers evenly across the entire network to avoid the situation where some hosts are underloaded while others are overloaded. Ferrari and Zhou, 1987 also discuss a similar architecture and describe the details of an early implementation. In Ferrari's system, every host can produce tasks. A software module is responsible for constantly exchanging load information with its peers on other hosts and performs job placements. If the local host is heavily loaded, while some other hosts are not,

one of the remote hosts is selected as the destination for the job.

The load balancing system tries to use the system state information to dispatch the workload within the system. The host load information across the system should be collected to aid decision-making. Which information and how to collect them are a part of the load balancing system. The scheduler will analyze system load information and make an evaluation of each host to choose an optimal one for the incoming task. In distributed systems, tasks may be scheduled centrally or in a distributed way. In a centralized approach, all tasks are sent to a scheduler that dispatches them. In the distributed approach, tasks are scheduled locally on the originating host which decides where the execution should be performed.

Considering the functions of the load balancing system, three basic policies are involved in load balancing (Shiraze and Hurson, 1995) (Chapter 5):

"

1. The information policy: specifies the amounts of load information made available to, and the way this information is distributed among, the job placement decision-makers. This information consists of:
   - Load information of all or a subset of hosts;
   - Load information is distributed periodically or on-demand at load balancing time;
   - The load information is collected at a central host or distributed among the participating job placement decision makers.
2. The transfer policy: determines the suitability of a process for task relocation; that is, the policy identifies whether or not a task is eligible to be transferred to another host. The transfer policy is based on several factors, including:

- host load – a process is eligible for transfer only if the load of its host is above some threshold value;
- process size – a process is eligible for transfer only if its size is above a threshold value;
- process resource needs – in a heterogeneous distributed processing system, a process may not be executable on a subset of the hosts or its execution time may vary considerably on different hosts.

3. The placement policy: determines the host to which a process should be transferred. the job placement decision-maker selects the target host for a transfer based on two general policies:
   - select the host with the minimum current load (minimum load);
   - select the host whose load is below some threshold value (low load)."

In the load balancing system, the task dispatching algorithms play a very important role in making a decision to select the destination host. There are many ways to classify the algorithms. Sender(source)- or receiver(server)-initiated classification is discussed by Wang and Morris, 1985. Supposing that tasks are created on sender (source) hosts and will be executed on receiver (server) hosts, "a load balancing algorithm is said to be sender- or source- initiated, if a local host (source node) makes a determination as to where a generated task or an arriving task is to be executed. In a receiver- or server- initiated load balancing algorithm, a server determines which jobs at different sources it will process" (Shiraze and Hurson, 1995) (Chapter 5). In the sender-initiated method, when a task is coming, a scheduler will check a group of candidate hosts and make an evaluation according to their load information to select the best one for executing the task. But in the receiver-initiated method, the destination hosts have the initiative. The receiver starts to apply for tasks when its load becomes light.

Another way of classifying load balancing algorithms is using their static or adaptive nature, "Static load balancing algorithms do not rely on the current state of the system at the time of decision-making. Random and round-robin placement policies are examples of static decision-making processes. Adaptive load balancing algorithms react to changes in the system state" (Shiraze and Hurson, 1995) (Chapter 5). The static algorithms distribute the load according to some rules that are set a priori. Static algorithms allocate servers permanently or semipermanently to arriving task streams. The major drawback of static algorithms is that they do not respond to short-term fluctuations of the workload. Dynamic schedules attempt to correct this drawback but are more difficult to implement and to analyze.

Depending on how load information is distributed and how processes are assigned to hosts, load balancing algorithms can be classified into central or distributed information and placement decision schemes (Shiraze and Hurson, 1995) (Chapter 5). In a central information and placement decision scheme, a central scheduler exists to collect load information from other hosts. When a job comes, the central scheduler selects one host using the load information it gathered. In distributed policy schemes, each host autonomously constructs its own local load vector by collecting load information from other hosts. When a task stems up from a host, it is scheduled by the host locally with the information stored in the local load vector. Mixed policy is implemented by a central scheduler collecting load information from other hosts and sending the load into other hosts periodically, so that each local host be able to make a decision with the updated load information from the central scheduler.

1.2 Algorithms

Load balancing algorithms are often classified into static and dynamic.

1.2.1 Static Algorithms

The canonical static algorithms include Random, Round-Robin, and Size-based assignment policies. In the Random task assignment policy, an incoming job is sent to host i with probability $1/h$, where h is the number of hosts. This policy equalizes the expected number of jobs at each host.

In Round-Robin assignment, tasks are assigned to hosts in a cyclical fashion. This policy equalizes the expected number of jobs at each host, and has slightly smaller variability in inter-arrival times than does Random.

Both the Random and Round-Robin policies use no system information at all. No exchange of state information among the hosts is required in deciding where to transfer a task. The implementation of these two algorithms is simple, but both can easily lead to a situation where some hosts are under heavy load, while some other hosts may be under a very light load, or even in an idle state. In their paper Wang and Morris, 1985 implement two algorithms under a distributed system. The first algorithm works as the source-initiated method, which selects the server hosts randomly and uniformly for coming tasks. The second algorithm belongs to the server-initiated category. When a server is available, it visits the source hosts at random and removes task(s) to run. The simulation experiments are taken under different task arrival distributions. The performance is analyzed with the queuing theory. Wang compares the performance with some other algorithms and indicates the drawbacks of the two algorithms coming from their insensitivity

to short-term fluctuations of the workload. Cao (Cao *et al.*, 2000) simulates the Random method under a workstation-based locally distributed system. In his system model, the coming tasks are waiting in a queue and the task scheduler assigns the tasks randomly to other workstations. Using a trace-driven approach, Cao exploits previous accounting files to generate the workload, and compares the system performance under different load states. It is observed that the Random algorithm can improve the system performance except if the entire system is in an overloaded state.

While load balancing systems offer potential performance improvement in the context of distributed system, one important issue that must be solved to realize good performance is the determination of an efficient schedule. However, effective schedules should integrate application-specific and system-specific information (Berman *et al.*, 1996). Users schedule tasks with application-specific information based on the knowledge of the structure of the system and requirement of the tasks. This knowledge generally includes the expected computation time, the memory requirement, the I/O state that is needed, and network communication, etc. The algorithm of Shortest Job First uses specific information about the jobs. This method can improve the performance of the entire system by decreasing the average waiting time for jobs. Size-based algorithms are other examples which use application information to decide task assignment. In the implementation of an application-specific system, both intuition and history information may be used as a base for schedule determination to predict the system behaviour.

The Size-Based policy assigns tasks according to the distribution of task sizes. The scheduler knows the information of the task size a priori, when a task is arriving, it is sent to the specific host according to this task information. In practical implementations, there are many variables that enter into the definition of the task

size. According to the type of the tasks, considered resource usage such as memory consumption, I/O activities and CPU time can be used to determine size. This idea has been discussed in papers by Harchol-Balter *et al.*, 1999; Schroeder and Harchol-Balter, 2000; and Crovella and Harchol-Balter, 1998. Size-based algorithms assign all tasks assuming that service demand is known. In their paper, Harchol-Balter *et al.*, 1999 introduce an algorithm called Size Interval Task Assignment with Equal Load (SITA-E) in which all tasks within a given range are assigned to a particular host. The idea is simple: "define the size range associated with each host such that the total workload directed to each host is the same" (Harchol-Balter *et al.*, 1999). "The mechanism for achieving balanced expected load at the hosts is to use the task size distribution to define the cutoff points (defining the ranges) so that the expected work directed to each host is the same. The task size distribution is easy to obtain by maintaining a histogram of all task sizes witnessed over a period of time"(Harchol-Balter *et al.*, 1999). SITA-E relies on the assumption that the distribution of the size of incoming requests is known. The SITA-E's performance remains remarkably insensitive to the increase in task size variability. In this algorithm, the most important thing is to collect the task sizes information and get the statistical analysis, then define the cutoff points. Crovella discusses another similar algorithm called Size Interval Task Assignment with Variable Load (SITA-V) (Crovella and Harchol-Balter, 1998) which is an adjustment of SITA-E. In the central scheduler, the task information (task size) is known and the assignment is completed based on it. The small tasks are sent to a group of hosts, while the large tasks are sent to another group. Through the control of the different loads in the different host groups, performance improvement can be achieved.

Devarakonda develops a statistical, pattern-recognition-based approach for predicting the CPU-time, the file-I/O, and the memory usage of a process at the beginning of its life, given the identity of the program being executed. "Initially, statistical

clustering is used to identify high-density regions of process resource usage for the measured system" (Devarakonda and Iyer, 1989). Devarakonda aims to predict the resource usage for each process and use the information to direct the execution of the process. The statistical data is analyzed and a state-transition model is built to represent the behavior of each process. Using the past resource usage information and the state-transition model, the future consumption of system resources can be predicted. Load balancing policies (Goswami *et al.*, 1989) are developed based on this resource usage.

The size-based algorithms need the task information known when the assignment is being performed. This requires the application developers to know their task characteristics correctly. Although a statistical method can help to gain this task information, the approach requires additional manual work for its implementation. Harchol-Balter brings out an algorithm to avoid this problem which doesn't need to know the task characteristics a priori (Harchol-Balter, 1999). The algorithm of Task Assignment based on Guessing Size (TAGS) (Harchol-Balter, 1999) does not require users to know the task size and does not collect system state information in task schedule. The task sizes information is hidden in the assignment policy and represented in the execution of the tasks. Based on Least-Work-Remaining policy, TAGS works as follows: "Think of the hosts as being numbered: 1,2,3...,h. The ith host has a number s_i associated with it, where $s_1 < s_2 < s_3 ... < s_h$. All incoming tasks are immediately dispatched to Host 1. There they are serviced in First Come First Serve (FCFS) order. If they complete before using up s1 amount of CPU, they simply leave the system. However, if a task has used s1 amount of CPU at Host 1 and still hasn't completed, then it is killed, the task is then put at the end of the queue at Host 2, where it is restarted form scratch. The TAGS algorithm does not require knowledge of job size" (Harchol-Balter, 1999). Comparing with the algorithm discussed by Harchol-Balter *et al.*, 1999, this method dispatches tasks

with different size ranges into different hosts with the execution information of the tasks instead of assigning them to fixed hosts by hand.

The size-based algorithms try to make the system load balanced using application information but without using dynamic system state information. The estimation of the application-specific information becomes a central component in the success of the algorithm. If the application size can not be assessed correctly, it can direct tasks into improper hosts and cause the system to be in an unbalanced state. The algorithm discussed in Harchol-Balter, 1999 can solve the problem of estimation for applications, but it will use up more system resources when tasks move from host s_i to s_{i+1}.

Our project is a MDO application which integrates many engineering analyses programs in different departments. Every engineering program has its own characteristics and costs for various system resources. "To obtain the best performance, the user must take into account both application-specific and dynamic system information in developing a schedule which meets his or her performance criteria" (Berman *et al.*, 1996). "One of the most fundamental problems that must be solved to realize good performance is the determination of an efficient schedule. Effective scheduling by the application developer or end-user involves the integration of application-specific and system-specific information" (Berman *et al.*, 1996). In our implementation of the Dynamic assignment algorithms, the application and system state will be considered to achieve good performance. At the same time, we compare different algorithms in our project and analyze their performance. Next we introduce briefly some relevant research work on these algorithms.

1.2.2 Dynamic Algorithms

To counter obvious problems found with static algorithms, many dynamic load balancing algorithms have been developed to utilize host information effectively, and evenly spread the workload across the network. In Dynamic assignment, an incoming task is assigned to the host with the lightest workload left to do. The question then becomes how to define and use the workload results in different implementations of load balancing algorithms. How to define the workload will be discussed in section 1.3.

In the central policy load balancing algorithms, the task enters the scheduler which makes a decision to choose one destination host on which the task is to execute. This class of algorithms can be found in papers such as (Harchol-Balter, 1999). In this algorithm model, a central scheduler exists to collect host load information across the network and evaluate the workload of the system to choose the best host (with the lightest load or above a threshold) for the execution of the task. In order to collect host load, a local collector should be installed on each host in the distributed system. Local collectors gather host load and communicate with the central scheduler to send their load value.

The core of the load balancing algorithm must evaluate host load and place tasks. One algorithm is called Threshold (Eager *et al.*, 1986). For host load information, every host has a threshold value, when the local load is above that value, the host is under a heavy load. Otherwise it is under a light load. When the dispatcher decides to assign a task, it selects a host under light load state for the new coming task. In their paper Eager *et al.*, 1986 give a detailed description of the algorithm. They indicate that Threshold is a location policy that acquires and uses a small amount of information about potential destination nodes. "Under this policy a

node is selected at random and probed to determine whether the transfer of a task to that node would place it above threshold (overloaded). If not, then the task is transferred; the destination node must process the task regardless of its state when the task actually arrives. If so, then another node is selected at random and probed in the same manner. This continues until either a suitable destination node is found, or the number of probes exceeds a static probe limit. In the latter case, the originating node must process the task" (Eager *et al.*, 1986).

The "best host" selection policy is used in many algorithms. When the scheduler needs to choose one destination host, it will evaluate all candidate hosts loads to select the least loaded one (Eager *et al.*, 1986; Svensson, 1990). An algorithm is described in Eager *et al.*, 1986 where a group of hosts are chosen at random, and each is polled in turn to determine its queue length. The host with the shortest queue length is selected and the task is transferred. Eager also compares the results between this algorithm and the Threshold algorithm. He indicates that the performance of the shortest queue length is not significantly better than that of the simpler threshold policy. Wang implements this algorithm under the name "Join the Shortest Queue" (JSQ). When a task comes, the scheduler checks every host and finds the server with the fewest number of tasks (Wang and Morris, 1985).

Wang and Morris, 1985 describe the server-initiated algorithm in some approaches such as: randomly receive tasks; receive tasks from the longest queue and receive the shortest task. In these server-initiated methods, the task's execution is decided by the local hosts. When the local host is lightly loaded or in an idle state, it sends a request to the sources or task dispatcher to get a task. The difference between this method and the source-initiated is that in the source-initiated method, the scheduling decision is made when the task is arriving, whereas in the server-initiated method, the scheduling decision is made when the task is departing. Wang

implements this algorithm under a multi-source and multi-server model. There are many source hosts from which tasks originate, and these tasks can be scheduled among a group of server hosts. Each source host maintains its task queue and waits to be scheduled. Serve the Longest Queue is described as: "When a host server becomes available or in an idle state, it requires a job from the longest source queue". Wang compares this method with the source-initiative method and concludes that this method is more sensitive to load distribution than the source-initiative algorithm. When the load becomes unbalanced in the system, the algorithm degrades because the lightly loaded queue receives low priority in gaining service.

The bidding algorithm combines the idea of source-initiative and server-initiative methods together. The method separates the decision role among the scheduler and server hosts. In the biding algorithm, with the arriving task, the scheduler sends a bid to a group of hosts. In the bid, some information about the task can be taken. The hosts which receive the bid will make a candidate evaluation for the bid according to their own state, and return the evaluation result to the dispatcher. When the scheduler gets the return information about the bid, it chooses one best host to assign the task (Rommel, 1991; Musliner and Boddy, 1996). Musliner implements the method by sending a bid into a group of hosts at task arrival. When the hosts receive the bid, they compute their current load state and make an evaluation of receiving the coming task. After comparing the two states they return a response to the dispatcher. The dispatcher then selects the best host from the proper candidates.

As we can see from the above algorithms, a central load balancing algorithm will have two important parts: Load information policy and schedule policy. The load information policy is responsible for transferring the host load information between

the host load collectors and scheduler. The schedule policy is used to assign tasks to the proper hosts. How to use the host load information is the function of the scheduler and generally application-specific and system-specific information will be used in the decision-making.

1.3 Load index

Load balancing systems try to send tasks in the network evenly to maintain each host resource at an adequate utilization level. During the task assignments period the current host load of the system should be estimated to decide which host is suitable for task execution. What information will be used and which information is more useful about the coming task bring us to the topic of load information. Load information is an integral part of dynamic load balancing algorithms. It is used by the load balancing algorithm to evaluate the entire system and then to make a decision on choosing a suitable host for the coming tasks. "In the load balancing algorithms, the workload has an important influence on the probability of load balancing success" (Rommel, 1991). "The efficiency of a load balancing scheme can be affected heavily by how the load information is chosen and when that information is updated" ([Chapter 7]Shiraze and Hurson, 1995). "Use of very small host information can achieve good performance in distributed system" (Eager *et al.*, 1986). Dynamic load balancing algorithms must collect and react to system state information, balance the system load among the distributed resources and utilize the potential system capacity effectively.

The load information is usually represented as a load index. This is used to indicate the host load which is used by the scheduler to estimate the host's situation. A wide variety of load information has been discussed (Ferrari and Zhou, 1987; Zhou,

1987; Kunz, 1991). Ferrari has discussed some simple load indexes in his paper (Ferrari and Zhou, 1987). The following load informations are compared:

1. the instantaneous CPU queue length;
2. exponentially averaged CPU queue length;
3. the sum of averaged CPU, file and paging/swapping I/O, and the memory queue lengths;
4. the average CPU utilization over a recent period.

Although use of different load indexes result in different system performance, they all work far better than no load balancing. The influence of system characteristics on the load index performance is also discussed. Ferrari mentions that in a specific application field and computing environment, the combination of a number of load indices constitutes a better index.

Since the load index plays an important role in the dynamic load balancing algorithms, its definition becomes an important problem that we must face. In their book Shiraze and Hurson, 1995 provide several characteristics of a good index:

"

- The load index should take into account not only the CPU needs of a process, but also its I/O and memory operation requirements;
- The load index must reflect quantitatively the qualitative estimates of the current load on a host;
- Since the response time of a job is affected more by the future load of a host than the present one, the load index should be usable for prediction of the near future load;

- The index should be relatively stable, that is, high frequency fluctuations in the load should be discounted or ignored;
- There needs to be a direct relationship between the load index and the performance of the system."

Some examples of load indices have been given in the book. Such as:

- CPU queue length: The load index is a function of the ready queue length. Some of the most effective load indices are defined as functions of several queue lengths, for example, average CPU, I/O, and memory queue lengths;
- CPU utilization: Some researchers think that a host's CPU utilization is a good indication of its load, and thus can be used as a load index;
- Response time or processing time: There are some choices that can be made. For example, summation of the remaining processing times of the jobs running on a host; summation of the processing times used by all the active processes up to the current time; summation of the total processing times of all the active processes;
- Aggregate functions: One can define the load index as a function of several of the above mentioned indices.

In the multi-disciplinary engineering application field, most jobs are CPU-bounded programs and those have a strong requirement on CPU demand, but some specialized tasks that are memory or I/O -bounded programs may also be encountered. For a task which needs more CPU computing, choosing a host with more CPU power is a good choice. Some large tasks also have stringent requirements in terms of memory, if the load index is only based on CPU, then the load of the host which

is chosen to execute the task is underestimated. The memory information should be considered in the schedule method. The same comment also applies to I/O capacity.

In a related study (Zhou, 1987; Kunz, 1991), it is shown that the process response time strongly depends on the processor load, and that the CPU and I/O queue lengths are good indicators of this load. Kunz carried out experiments on many kinds of tasks with varying attributes (CPU-bound, I/O-bound, tasks balancing the CPU and I/O, etc) and analyzed the result of CPU and memory load indices. The following single workload descriptors were discussed:

1. number of tasks in the run queue;
2. size of the free available memory;
3. rate of CPU context switches;
4. rate of system calls;
5. 1-min load average;
6. the amount of free CPU time.

Kunz concludes that all the load indices obtain a better system performance than the situation where no load information is used.

The memory utilization is important to define the load index. During the execution of the task, frequent swap-in and swap-out of pages between memory and disk will degrade the performance of the system heavily (Xiao *et al.*, 2000; Berman *et al.*, 1996). In the assignment of a task, the memory of the destination host has a basic influence on the performance in the host. Xiao has mentioned a measure involving

the cooperation of CPU and memory (Xiao *et al.*, 2000). Xiao brings out a load index which considers the CPU and memory resources. The idea of Xiao's measure is that when a task needs to be assigned, the candidate host should be checked for the memory resource first, when the memory resource is suitable for the waiting task, a host with the most powerful CPU becomes the best choice. In this method, large number of page faults are avoided. Otherwise, the job may be delayed in the host because of insufficient memory. Xiao concludes that the CPU and memory method can work well in a heterogeneous system as compared to the CPU only load index. The CPU-based method could easily send a task to a host with limited memory as opposed to the CPU and memory method which can identify hosts that lack sufficient memory to run a task. In this way the CPU and memory can offset the heterogeneity effects of the system.

The combined use of CPU, memory and I/O are discussed in papers (Zhou, 1987; Devarakonda and Iyer, 1989; Ferrari and Zhou, 1987). Devarakonda presents how to get CPU, memory and I/O information for a task during its execution, and uses these three resource-usage parameters in load balancing algorithms. Devarakonda uses a statistical pattern to collect these resources information, and applies these resources information to the task's assignment. Devarakonda does not mention the effectiveness of the approach in his paper (Devarakonda and Iyer, 1989), but these resources are used in a centralized load-sharing method (Goswami *et al.*, 1989) and they result in a good performance improvement. The advantage of the combination of multi-resources in a multi-disciplinary application is obvious. In an I/O-bound task, the CPU requirement is not high. If only the CPU is used as the work index, the assignment of the task is underestimated. In this case, the I/O information of the system should be considered and applied in the decision-making procedure. Ferrari implements the index combining CPU, memory and I/O. This index works well and gives an improvement to the non load balancing situation (Ferrari and

Zhou, 1987). A linear combination of resource queue lengths was proposed as a load index. In that linear combination, the coefficient of a resource queue length is the amount of service time that the particular job being considered requires from that resource. The index is response time oriented and job dependent. "Instead of a unique value at a particular moment in time, the load of a host differs for different jobs because of their varying resource demands, which are assumed to be known upon job arrival" (Ferrari and Zhou, 1987). In this policy, the application-specific information and system-specific information are used together to make assignment decisions. Another kind of combination of resources are also studied there in which the coefficient of the resource queue lengths are *job independent*, and only reflect the relative importance of the resources. In the experiments, all the indices provide performance improvement, and another result is that the performance of load balancing is heavily dependent on the load index used. By comparing the different load indices, Ferrari indicates that in a system, to achieve near-optimal performance, it is not necessary to consider all the system resources, only those with significant contention. Xiao *et al.*, 2000 supply another algorithm which uses the CPU, memory and I/O together. Xiao expresses the load index such that the requested memory allocation should be less than the available memory in the current system. If this condition is satisfied, then he tries to find a lightly loaded CPU. The I/O requirements of tasks are treated as a factor of CPU load.

In a homogeneous system, the system resource utilization can reflect the system load. But in a heterogeneous distributed system, the system capacities should be used in the system load index. In his paper Ezzat, 1986 includes the average CPU utilization and the CPU capacity at a host in his definition of a load index. "Let C_j denote the CPU capacity of host j. If $C_j = 2C_i$, and the two hosts have the same load, then the expected response time of host j is half that of host i"(Ezzat, 1986). Mehra and Wah, 1993 discuss the fact that the existing load indices which

only consider the system resource utilization (especially only CPU resources) can not precisely represent the host load and will fail when the distributed system is heterogeneous. CPU power and memory capacities in a network of workstations are discussed in Xiao *et al.*, 2000 to express system heterogeneity, and used as load index. The CPU load index in each node is calculated by

$$L_j \times \frac{1}{(W_{cpu}(j))}, j = 1, ..., P, \tag{1.1}$$

where L_j is the number of jobs queued in node j, $W_{cpu}(j)$ is the CPU weight of the node, and P is the total number of nodes in the system. The $W_{cpu}(j)$ is defined as:

$$W_{cpu}(j) = \frac{V_{cpu}(j)}{max_{i=1}^{P} V_{cpu}(i)}, \tag{1.2}$$

where the $V_{cpu}(j)$ is the speed of workstation M_j, j=1,...,P.

The power and memory capacities are used in load balancing algorithms (Xiao *et al.*, 2000; Xiao *et al.*, 2002). The CPU threshold is set based on the CPU computing capacity and used to indicate if the host CPU is overloaded. The available memory is used to limit the memory allocation for an arriving task.

1.4 Load information update

The estimation and updating of a host's load information are among the issues that need to be addressed in dynamic load balancing. These issues are of particular

importance because the load information serves as one of the most fundamental elements in the load balancing process. In the source-initiative algorithms, the scheduler needs to know other host loads in the system. There are three ways to get this system information: demand-driven, periodic collection and state change driven (Lu and Liu, 1996).

In the demand-driven method, the scheduler sends the request to a set of hosts to get the information. The destination hosts receive this request and collect their local host load. Then they send the load information back to the scheduler. This method requires the scheduler to collect the host loads frequently, which makes the scheduler spend significant amounts of time waiting for the responses of the destination hosts. The delay of system state collection degrades the system performance when the dispatcher needs to deal with frequent incoming tasks.

The periodic information collection sets up local collectors to periodically gain host load information (Mitzenmacher, 1997; Ferrari and Zhou, 1987; Ezzat, 1986). Ezzat (Ezzat, 1986) describes the algorithm as follows: At every T_s units of time, local hosts collect their load information. This routine is repeated n times at every T_s interval. The n load values are averaged and sent on the network. Every T_u time units, the information policy module calculates the averaged load information for the most recent period of time and sends it on the network. In this method, the parameters n, T_s and T_u depend on specific application characteristics. This scheme can avoid the instantaneous value of the system state. In algorithms where the host state is updated periodically, the interval of updating state information must be correlated with task arrival frequency. For a system with frequent arriving tasks, the interval of updating state should be short, otherwise, the collected information can not represent the real load of the host, it becomes stale. If the interval is too short, some emerging peak load will influence the precise estimation of the host

load.

In the state change driven method, the local host states are checked periodically. If the new state information has a large difference with the currently held value, the new state information is sent back to the scheduler. Otherwise, the new value is thrown away. Ferrari talks about this method in his paper (Ferrari and Zhou, 1987). His system includes a local information module in each local host. Every time period P, the local host extracts its load information to compute the local host's index. If the new value of the load index is significantly different from the previous one, the new value is sent to the master information module which collects load information from every host. The performance of this method is influenced by the parameter P (period value). If the exchange value is very short, the load information is up to date, but its effectiveness is influenced by the high message overhead. If the period P is very long, the information can not reflect the rapid change of system state. The performance is degraded. This parameter is relevant in the context of a specific applications. If task arrival is frequent, the period time should be short.

As indicated in the periodic algorithm, the history information is used to avoid influence of peaks on the host load. In most load balancing algorithms, if the load information is too old and does not reflect the system load correctly, system performance can easily degrade. Mitzenmacher (Mitzenmacher, 1997) discusses this topic using a theoretical system model to analyze the influence of performance with the interval time T (T_u in (Ezzat, 1986); P in (Ferrari and Zhou, 1987)) used in the load information update. Load information is updated in interval time T, the old information is used in host evaluations. Based on those simulation results, the old information with the optimal chosen interval time T can be used to improve the system performance in a load balancing system . The shortest queue and random

algorithms are selected to compare with the algorithm which selects the lightly loaded host from the randomly selected two hosts (2 chosen). When T is small, the *shortest queue* and *2 chosen* algorithms obtain a better system performance than the random algorithm. But as T grows larger, the random algorithm performs better. "if the update interval T is sufficiently small, so that only a few new tasks arrive every T seconds, then choosing a shortest queue performs very well, as tasks tend to wait at servers with short queues. As T grows larger, a problem arises; all the tasks that arrive over those T seconds will go only to the small set of servers that appear lightly loaded, overloading them while other servers empty"(Mitzenmacher, 1997).

The periodic information collection looks like the way which approaches the VADOR system.

1.5 Task classification

The performance of a system is determined by its characteristics as well as by the composition of the tasks being processed (Calzarossa and Serazzi, 1993). The most popular method adopted for workload characterization of batch and interactive systems is the clustering. The significative clusters are characterized by a set of parameters such as CPU, memory and number of I/O accesses. "The resource level represents the system point of view: how the systems resources (CPU, memory, I/O bandwidth) are being used"(Pasquale *et al.*, 1991). At the resource level, jobs are classified according to resource consumption statistics, especially CPU time, I/O channel time, and memory space-time product. Based on the amounts and relative percentages of system resources consumption seven clusters are listed out. In his paper Kotsis, 1997 presents a model of CPU, memory and I/O resources

consumption using the Kiviat diagrams. By comparing the computation, memory and I/O demands, a program can be characterized as being either computation, memory or I/O bound. If a program spends excessive elapsed time managing memory, it is considered a memory-bound program. If a program spends most of its elapsed time performing I/O, it is considered I/O bound. CPU-bound programs spend most of their elapsed time performing calculations on the processor.

In their paper, Pasquale and Polyzos, 1993 consider that I/O intensive workload is characterized by extensive demands in terms of I/O volume, i.e., total number of bytes transferred, and average *virtual I/O rate*, i.e., number of bytes transferred per CPU second. Focusing more on I/O rates rather than total I/O volume is dictated to investigate applications that might stress the I/O system to the point of affecting their response time or interfering with other applications. In their investigation of I/O intensive jobs, only the jobs which exert a long, sustained demand on system resources are considered. For example, in their working environment of the San Diego Supercomputer Center Cray Y-MP8/864, the I/O intensive jobs are chosen with average CPU time exceeding 100 seconds and the I/O rate of the jobs is at least 3.31 MB/cpu second.

Several large scientific and system programs are selected as representative CPU-intensive, memory-intensive or I/O-active jobs in (Xiao *et al.*, 2002). *Bit-reversals* is a program which conducts data reordering operations which are required in Fast Fourier Transform algorithms and this is CPU-intensive and memory-intensive. *Matrix multiplication* is a standard matrix multiplication program and is both CPU-intensive and memory-intensive. In their experiment environment, the memory-intensive programs have about 60 MB memory requirement in a system environment with 128MB of main memory. Some SPEC 2000 benchmark programs are also used in this paper (Xiao *et al.*, 2002), such as *gzip, mcf, vortex, and bzip* which

belong to CPU and memory intensive program classes.

1.6 Thresholds

Threshold values are used in the load balancing algorithms to distinguish host load states. Eager (Eager *et al.*, 1986) uses a threshold to decide if an assignment of a task causes the destination host to reach a value which would bring the host in a heavy load. "Each host periodically broadcasts its status (underloaded or overloaded) to all other hosts. To be able to determine this status, the value of a specified workload descriptor is measured and compared to a given threshold. A host is considered overloaded when the measured workload exceeds the threshold. Determining the optimal threshold value is not trivial and depends on the network wide load"(Kunz, 1991). Multiple thresholds can be used to determine the different load states, such as light, middle and heavy that use 2 thresholds. Since host load can be represented by various load indices, the definition of thresholds should be determined in conjunction with the choice of load index.

Kunz(Kunz, 1991) tests the percentage of idle CPU-time as the index and gets an optimal value of 2% in his experiments. He also indicates that the percentage of idle CPU-time is influenced by the status update period length used. In their book Johnson *et al.*, 1999, a CPU-bound system is defined as a system where the percentage of idle CPU-time is less then 5%. In the paper by Ferrari and Zhou, 1987, CPU utilization is used to define the workload levels. Light workload is defined as a 30-45% CPU utilization and the heavy workload is defined as 70-85%. Xiao (Xiao *et al.*, 2002) uses the CPU capacity to set the CPU threshold and the available memory to define the memory threshold. The idea of the memory utilization is that the memory space allocated for the active processes should be

less then the available memory to avoid the frequent page in and page out. Another kind of definition of threshold for memory is the amount of free memory in the host discussed by Ferrari and Zhou, 1987.

1.7 Measure of the performance

Load balancing algorithms aim to improve the system performance. How to measure the system performance is a problem that must be faced when the load balancing algorithms are implemented. Different measurement standards have a big influence on the estimation of the algorithm efficiency. It is also a considerable factor in the selection of an algorithm. The two general approaches to performance measurement are *mean response time* and *mean slowdown*.

The mean response time measures the throughput of the entire system, it represents the delay of the task in the system. A task's slowdown is its response time divided by its service requirement. Mean slowdown is important because it is desirable that a task's delay should be proportional to its processing requirement (Harchol-Balter, 1999). Generally, short tasks are expected to have short waiting time, but large tasks can tolerate a longer delay.

Another performance goal, *fairness*, is used in some load balancing systems. That is to say that all jobs, long or short, should experience the same expected slowdown. In particular, long jobs should not be penalized (Schroeder and Harchol-Balter, 2000; Harchol-Balter, 1999).

1.8 Load balancing system and products

Load balancing algorithms are integrated into a large set of real applications, from distributed computer chess, to finite element methods, Web application, and CFD optimization. Some load balancing systems have been developed for utilization in specific application fields. Following is a brief survey of widely used and available products that address load balancing issues.

1.8.1 OpenPBS

This system (Portable Batch System) aims to provide additional controls over initiating or scheduling execution of batch jobs; and to allow routing of those jobs between different hosts. It allows a site to define and implement policies as to what types of resources and how much of each resource can be used by different jobs. This system provides transparent job scheduling on any system by any authorized user, it provides numerous ways to distribute the workload across a cluster of machines.

The batch system is made up of a number of components. The server, scheduler and resource monitor. The server manages batch objects such as queues and jobs. it provides batch services like creating, routing, executing, modifying or deleting jobs for batch clients. The scheduler is used to decide which jobs should be placed into execution. The resource monitor is used to execute the jobs and monitor the resource utilization. The scheduler receives commands from the batch server (which receives commands from clients) and communicates with the resource monitors to get the system resource information.

Seven load balancing algorithms are implemented in the scheduler. A central load balancing architecture is used in this system. The *1 minute average load* is collected

as the load index.

1.8.2 BALANCE

The BALANCE system is a flexible, computer architecture independent and network independent load balancing system which is designed to support a wide range of software (Hui *et al.*, 1995). It is not tied to a particular scheduling algorithm, rather the users are allowed to build their own schedulers.

Local and global load servers are implemented in the system. The local server collects workload information on the local computer, and the global server gathers, sorts and maintains the loading information provided by the local servers. Several load balancing algorithms are developed in the system.

1. Immediate scheduling algorithms: This is a centralized dynamic scheduling algorithm which sends the coming tasks to the least loaded host. In this algorithm, the load index is the *run queue length* (the number of tasks in the run queue).

2. Random scheduling algorithms: In this method, the scheduler dispatches the tasks to run on hosts in the system in random order.

3. Delay scheduling algorithms: This is a modification method for other algorithms and is responsible for handling the situation with the system in high load. The basic principle is to stop sending new jobs out when the system loading is higher than a threshold.

1.8.3 lbnamed

Lbnamed is a load balancing name server written in Perl. It allows to create dynamic groups of hosts that have one name in the DNS name space. The architecture of the system includes three parts: *lbnamed, poller and client daemon.* The *client daemon* is running on the hosts to collect the local host load. The *poller* works as a global collector, it contacts the *client daemons* by periodically sending out requests and receives the responses asynchronously. After it has received all the responses it saves them into a configuration file and sends a signal to *lbnamed. Lbnamed* reads the host load information gathered by the *poller* and saves it in its inner structure. When a request comes for a host in a group of hosts, the best host is selected as the destination one, and the load value is slightly increased.

In the definition of load index, the load average over the last minute is used combined with the number of users logged in.

1.8.4 LSF

Platform LSF (Load Sharing Facility) is a commercial workload management solution that optimizes the use of enterprise-wide resources by providing transparent, on demand access to computing resources. It ensures fair sharing of resources among users and provides information about the state of a computing environment.

Platform LSF ensures optimal server utilization by continually monitoring server resources. The system base consists of two network servers, LIM and RES. The Load Information Manager (LIM) runs on every host, makes job placement decisions and provides built-in load indices about CPU, memory, swap, /tmp, I/O traffic, active users and interactive idle time. The Remote Execution Server (RES)

carries out job execution requests of the LIM on all networked hosts.

LSF also supplies a central queue to dispatch the jobs with the information gathered in LIM. In this system, the resource requirements are used in the decision with the host load. The resource requirements of a job are expressed as a resource requirement expression. When placing jobs, LSF matches the resource requirements expression against the available resources of the cluster.

1.8.5 Others tools

In parallel with the development of distributed systems, load balancing algorithms have been implemented in many Job Management Systems and Web applications.

Codine is a software package targeted at utilizing heterogeneous networked environments, in particular large workstation clusters with integrated compute servers, like vector and parallel computers. Codine also provides dynamic and static load balancing.

MOSIX is a software package for Linux that transforms independent Linux machines into a cluster that works like a single system and performs load balancing for a particular process across the nodes of the cluster. MOSIX's algorithms are designed to respond to variations in the resource usage among the nodes by migrating processes from one node to another, preemptively and transparently, for load balancing and to prevent memory exhaustion at any node.

YALB (Yet Another Load Balancing System) is a load balancing system for heterogeneous workstation environments. The CPU-runqueue and the speed factor of the machine are collected in the local host and sent to the Load-Manager which collects the local load from all hosts and selects the fastest for execution.

Load balancing algorithms improve the system performance in distributed systems. The VADOR project is an application framework which integrates various software resources and allows them to work together. In the VADOR architecture (which will be discussed in the next chapter), tasks are created at different workstations and sent into a central management workstation to make schedules there. The load balancing system is introduced into the VADOR project with a central management model consistent with VADOR's characteristics. In the load balancing system, the central management module is used to collect system load from each host in the network. A local collector is resident on each host to gather host load information.

The VADOR project is targeted towards the MDO application field, where most application softwares belong to scientific or engineering computing, and some of them are memory intensive or I/O active jobs. For that reason, CPU, memory and I/O system resources must be considered in the load balancing system. Some load balancing system or tools are suitable for our architecture but have limitations regarding the type of informations used for scheduling. For instance, most systems, such as **OpenPBS** and **lbnamed** only provide a *1-minute average* load as index. **BALANCE** uses the *run queue length* as the index and does not mention the memory resource utilization. **LSF** supplies many load indices such as CPU, load average and memory. But these indices are not flexible enough to satisfy our project requirements. For instance, only the last minute CPU utilization is defined as CPU load. Similar problems exist in other tools, **MOSIX** is a Linux-based software package and difficult to expand to heterogeneous system; **YALB** only supports CPU state. **Codine** aims to be used in heterogeneous environment, but it doesn't support all the platforms and will influence the utilization of the VADOR project.

Based on the specific requirement of the VADOR project and the shortcomings

of these existing load balancing systems, we elected to design and develop the load balancing system by ourselves and make it an integral part of the VADOR framework.

CHAPTER 2

LOAD INDEXES AND NODE CALIBRATION

In a load balancing system, host load information is used to help assign tasks across the network. Host load indexes are quantitative values which try to express the host load. Based on the discussion of § 1.3, this chapter presents the actual load indexes used in the VADOR framework, and discusses the means used to account for node heterogeneity in the definition of the indexes.

2.1 Host load indexes

There are many indexes that can be used to estimate host load. For instance, user CPU usage, system CPU usage, CPU idle; current free memory, context switch; system buffer activities, page fault, and so on. In our load balancing system, we consider all CPU times, memory and I/O information to schedule tasks assignment.

2.1.1 CPU load indexes

Concerning CPU load, the percentage of idle CPU is selected to be the index. A high idle percentage means that the CPU stays in idle state a significant fraction of the time and that the CPU process power is not used efficiently. A low idle percentage means that the CPU is busy in handling system transactions. The CPU load index can be expressed as follows:

The CPU load:

$$L_{cpu} = I \tag{2.1}$$

where I is the percentage of idle CPU-time.

In a distributed system with an heterogeneous architecture, the CPU idle percentage may be insufficient. Hosts have different configurations, some hosts have powerful CPU capacity while others have lower power. CPU capacity must then be taken into account. This is also true of the memory and I/O subsystems; § 2.2 discusses in details approaches to calibrate nodes and the approach we have chosen to quantify node capacity.

Let I express the CPU idle percentage, C_{cpu} expresses the CPU capacity, the CPU load L_{cpu} in a heterogeneous system could be expressed as :

$$L_{cpu} = I \times C_{cpu} \tag{2.2}$$

How to calibrate the host CPU capacity is detailed in § 2.2.1 and § 2.3.1.

2.1.2 Memory load indexes

We select the current free memory of the host as the memory index in our Dynamic load balancing algorithm. If a host has large free memory, the opportunity of content exchange between memory and storage media becomes small with an incoming task. Otherwise, frequent memory page switches will obviously degrade the system performance. If a host has large free memory, we can say that the memory load

is light. If a host has small free memory, its memory load is heavy. The memory load index is thus expressed as:

$$L_{mem} = M_{free} \tag{2.3}$$

where M_{free} is the amount of free memory.

In an heterogeneous system, memory configuration has a big influence on host power. Hosts with large physical memory could run more processes without running into a problem of lack of memory. The memory size thus becomes an important factor in system performance. Again, in a heterogeneous environment, variations among nodes must be taken into account.

Let M_{free} express the amount of the free memory in a host, and C_{mem} represent the memory capacity in a host, the host load of memory L_{mem} can be defined as:

$$L_{mem} = M_{free} \times C_{mem} \tag{2.4}$$

§ 2.2.2 and § 2.3.2 present in detail how to calibrate the host memory capacity.

2.1.3 I/O load indexes

The I/O subsystem is responsible for transferring data between software and hardware. As a process runs, data is read and written from memory to disk. Process code and data need to be read from disk, and the execution results are written in file(s) which reside on the hardware system.

Traditional UNIX-based kernels create and maintain a set of buffers called the "system buffers"or "system buffer cache". The cache is a pool of memory located in the kernel's memory space. In more recent UNIX-based systems, the buffer cache space is allocated dynamically as needed. When a user issues a system-buffered write request, the user's data is simply copied to the correct position in the buffers in the system cache. The user's program waits for completion of the write request (unless an asynchronous operation was requested) until the requested data in the user's data area is completely copied to the buffers. Buffered data remains in the cache until the page is needed. An automatic OS process then transfers cache control to disk. "Dirty" buffers become "clean" buffers when they are synchronized with disk storage.

The buffered I/O between disk and memory is used to indicate if the I/O system is in a heavy load state or not. The buffer activities between memory and disk are chosen as the load index. It includes I/O reads and writes numbers. Without concerns for system heterogeneity, the I/O load index can be defined as:

$$L_{io} = \frac{1}{N_{io}} \tag{2.5}$$

where N_{io} is the number of buffer reads/writes between system buffer and disks.

In an heterogeneous system, the potential I/O capacity of hosts varies widely. In the running system, hosts with strong I/O system can perform more I/O operations simultaneously without degrading the host's performance. For an I/O-bound task, assigning it to this kind of hosts can improve system efficiency.

Let N_{io} express the numbers of I/O reads and writes, C_{io} expresses the I/O system capacity, the I/O load index can be defined as:

$$L_{io} = \frac{C_{io}}{N_{io}} \tag{2.6}$$

The calibrations of I/O system capacity are presented in § 2.2.3 and § 2.3.3.

2.2 Node calibration

Since one of the main goals of the VADOR project is to deal with large scale distributed computing environments in industrial settings, heterogeneity of the execution environment must be addressed from the start. One of the main consequences of system heterogeneity on load balancing is the large discrepancies among host capacities and performance that can be observed. These discrepancies may have a significant impact on the validity of the decision making algorithms that can be used to distribute load, and one of the main objectives of this research project is to investigate them.

This section aims to quantitatively estimate the capacity of the various nodes that

constitute the execution environment, as this capacity may be accounted for by some of the load balancing algorithms. CPU, memory and I/O systems are three main parts of a computer. Each of them can influence the computer performance significantly. Approaches to measure the capacity of the CPU, memory and I/O systems are discussed in this section.

2.2.1 CPU performance

There are two common ways of reporting the CPU performance of a computer. The first is to use the system parameters. The clock rate of the processor (usually measured in *MHz*) is often directly used as a measure of CPU power. For a single architecture, the *MHz* rating is always available and effective for comparing processor speeds. However, this is not accurate in comparing different architectures. "Instead of measuring computer central processing units by the megahertz of clock speeds, AMD is launching an initiative to develop a new reliable metric to judge CPU performance in standard personal computer applications" (Sellers, 2001). AMD's proposal for setting new standards for measuring processors is called the "True Performance Initiative"(TPI). AMD promised that TPI would be an industry-wide initiative to develop a new and more complete measure of performance that end-users could trust (Wasson, 2002). But until now, TPI has not been used widely and is not yet convenient for comparing with other platforms. MIPS (Million Instructions Per Second) was a popular method for a while in measuring CPU performance. "The MIPS rating of a CPU refers to how many low-level machine code instructions a processor can execute in one second" (Mipscpu, 2003). Unfortunately, using this number as a way of measuring processor performance is pointless because of the very large diversity of processors working under very different architectures. The MIPS figure became almost useless as soon as the Reduced

Instruction Set Computers (RISC) processors became popular.

Another way to measure performance is to rely on benchmarks. These benchmarks are programs which are developed in an attempt to mimic the behavior of existing applications. In order to be a useful CPU test, benchmark applications would be CPU-bound running calculations for the measurement interval. Aside from the problem of making codes truly representative of real applications, benchmarks began to fall a foul of the improvements that were made in compiler optimization. These improved compilers could determine that many computations were not actually being used and optimized them out of code, making a mockery of the benchmark (Sharp and Bacon, 2001).

Realizing that a realistic and widely used benchmark would be a major step forward, a group of companies, including DEC, HP, IBM, Intel, and Sun, joined together to form SPEC (Standard Performance Evaluation Corporation). The SPEC is a nonprofit consortium whose members include hardware vendors, software vendors, universities, customers, and consultants. SPEC's mission is to develop technically credible and objective component and system-level benchmarks for multiple operating systems and environments. "SPEC has identified a set of programs in widespread use, frozen the source code, established a way to measure performance, and defined a formula for averaging the individual results" (Sharp and Bacon, 2001). The SPEC measures the performance of each program and combines the values into summary statistics. The method to measure a program is to time its execution and compute the SPECratio by dividing a reference value by the execution time. The reference value generally is the execution time on a specific machine. SPEC defined two summary metrics: SPECint and SPECfp. To compute SPECint, the benchmark finds the geometric average of the SPECratios for each integer-based program. SPECfp is the analogous result for the floating-point

programs. SPEC used a group of applications to measure the CPU performance and has insured large diversity of the applications to prevent manufacturers from optimizing their machines for the set of applications. "Although manufacturers examine each SPEC program carefully and tune a machine to improve its rating, the size and diversity of the SPEC applications make it difficult to perform them well without also speeding up everyone else's code" (Sharp and Bacon, 2001).

We use SPEC results as the CPU power descriptions. SPECint and SPECfp are used to measure the integer and floating point processing power of a processor. Which one should be chosen to measure the CPU capacity should be determined by the workloads used in our applications. The programs which are used by our workloads in our experimental environment are introduced in § 4.2. Among the four CPU-bound programs (Queens, Pi, Optim-Nurbs and SumTwist), *Queens* and *Pi* programs use more CPU resources than *Optim-Nurbs* and *SumTwist*. The CPU-bound workloads are thus dominated by *Queens* and *Pi* programs. Because the *Queens* and *Pi* are integer-based programs, the SPECint is selected to express the CPU capacity.

2.2.2 Memory capacity

The memory subsystem plays a very important role in the system performance. Having a high speed CPU can not yield good system performance without getting the data in and out from memory quickly enough to keep the CPU busy (MemmanageHP, 2003; Memsize, 2003; McCalpin, 1995). The memory subsystem performance relies on many factors, such as: hardware speed, interface with CPU, BUS speed, memory size, etc.

Just like CPUs, the performance of the memory subsystem relies on the clock speed

and parallelism. Memory bandwidth is an obvious candidate for the performance evaluation. The architectural factors which determine the memory bandwidth are complex. Generally, the information on memory bandwidth is not typically available from published vendor data. Benchmark programs are developed to measure memory bandwidth. The *stream* is a synthetic benchmark, written in standard Fortran 77, which measures the performance of four long vector operations and is used to evaluate various systems (McCalpin, 1995).

The memory is used to store text and data which is needed to execute a program. The development of multitasking systems relies heavily on the use of virtual memory to manage limitations in memory size. When the amount of virtual memory in use greatly exceeds the amount of real memory, the operating system will spend a lot of time swapping pages between memory and disk. Because the disk speed is much lower than the memory speed, frequent page ins/outs greatly degrade system performance. Larger memory size can let more processes run in physical memory without exchange of contents with disk(s). Comparing two systems with different memory sizes, the machine with more memory can process more tasks simultaneously with good performance than the machine with less. From this standpoint, the memory size influences the memory performance.

2.2.3 I/O capacity

As processors continue to improve their performance faster than I/O devices, I/O increasingly becomes the system bottleneck. There is therefore an increased need to understand and compare the performance of I/O systems (Chen and Patterson, 1993).

The performance of the I/O system can be influenced by many factors, such as

speed of disk(s), bandwidth of disk interface, size of system buffer, speed of data bus, etc. "The hardware determines potential I/O performance, but the operating system determines how much of that potential is delivered"(Chen and Patterson, 1994). Since the I/O capacity is determined by software and hardware, it is not easy to assess the I/O capacity with part or all of these elements. Data transfer rate is a good way to evaluate the I/O system performance. But in a heterogeneous system, it is a big job to find the vendor of the hardware of the I/O systems and search the standard performance marks. Based on application characteristics, many benchmark programs have been developed to measure the I/O performance of a computer system.

I/O benchmarks focus on measuring the bandwidth of data transfers between memory and disk. This measurement generally is done through reading and writing data on disk(s) and evaluating how much time elapses. Many earlier versions of benchmark programs have limitations on their utilization, such as non I/O-limited, weak scaling strategy and narrow application range. These shortcomings have been discussed by Chen and Patterson, 1993. Rabenseifner and Koniges, 2001 supply a benchmark which includes many parameters for various system configurations, such as test date size, process number and file system. This benchmark is suitable to measure various systems, but it is very complex to measure a fairly simple problem. Furthermore, this benchmark is based on the MPI standard and focuses on parallel applications, it does not fit our test environment very well. Lmbench provides a suite of benchmarks that attempt to measure the most commonly found performance bottlenecks in a wide range of system applications (McVoy and Staelin, 1996). Lmbench includes a small and simple benchmark, *lmdd* that measures disk and file I/O. lmdd can take both special devices and raw devices as its input and output. On completion, lmdd outputs throughput and timing information instead of outputting the number of blocks transferred. However, lmdd is not well suited

to copying data contained on large disk(s). *Bonnie* is a benchmark written to explore disk I/O throughput. Bonnie performs a series of tests on a file of know size by reading and writing from/to disk(s). Bonnie tests the speed of file I/O using standard C library calls. This benchmark has been used popularly for several years.

Many I/O benchmarks have been developed for various applications characteristics. We prefer to test our system with a simple method which still reflects our application characteristics. In our workloads, most I/O-bound tasks are simply to output character strings and store them in a data file. Based on this situation, simply moving data between memory and disk, a program is simulated to test I/O system based on the idea used in most benchmarks. We use the Unix *cp* command to estimate the time needed to copy files for approximating the performance figures of the respective I/O systems. We test host I/O system capacity by copying a large file and accessing the time this copying command elapsed. The same operation was performed many times and all response times were averaged.

The data size directly influences test results. If the size is too small data completely fits in cache, and the performance may then become very good. In this situation, the test program does not read data from disk for the other test loops. Accurate results can be obtained based on different data sizes. In our system, the bandwidth of the I/O system is measured according to the following formula.

$$B_{io} = \frac{Datasizes}{ElapsedTimes} \tag{2.7}$$

2.3 Node calibrations of a heterogeneous environment

Our experiments are carried out under two different environments. One is a homogeneous network and the other one is a heterogeneous network. Table 2.1 presents the configuration of the homogeneous environment.

Table 2.1 Node configuration (homogeneous network)

Number of nodes	O.S.	CPU type	CPU speed (HZ)	Memory (MB)
4	Linux kernel 2.9.7	AMD x86	1 * 1.3G	1024

In the heterogeneous environment, six hosts are connected to form a distributed system across a local area network. The six hosts have the following configurations (Table 2.2).

Table 2.2 Node configuration (heterogeneous network)

Host No.	O.S.	FPU	CPU type	CPU speed (MHZ)	Memory (MB)	secondary/ Instruction/ Data cache (KB)
host1	IRIX65	R10010	R10000	1 * 195 IP28	512	1024 + 32 + 32
host2	IRIX65	R10010	R10000	1 * 195 IP28	256	1024 + 32 + 32
host3	IRIX65	R10010	R10000	1 * 175 IP28	256	1024 + 32 + 32
host4	IRIX65	R10010	R10000	1 * 195 IP28	256	1024 + 32 + 32
host5	IRIX65	R12010	R12000	2 * 400 IP30	512	2048 + 32 + 32
host6	IRIX65	R10010	R10000	1 * 175 IP28	256	1024 + 32 + 32

Note: All FPUs and CPUs are from *MIPS*.

In the homogeneous network, each host has the same capacity, it is not necessary to calibrate their capacities. The following work aims to calibrate the heterogeneous network.

2.3.1 Calibration of CPU capacity

The newest version of SPECint is SPEC2000. However, the SPEC2000 values for the hosts in our test environment are not published. Based on the host configuration of our test environment, the SPECint95 is used to express the CPU capacity (DiMarco, 1997). In our system, the percentage idle of CPU-time is used as the CPU load index and this value ranges from 0 to 100. Most of the SPECint95 values for our hosts are below 10. In order to deal with these values more conveniently, we multiply them by 10 and convert them into integer. According to the CPU load formula expressed above, this enlargement does not influence the comparisons of CPU load values.

The six host CPU capacities are listed in table 2.6.

2.3.2 Calibration of memory capacity

In our experimental environment, the system has the same memory architecture, the size plays an important role in the memory performance. In our test workloads, multitasking would be the main characteristic in the system execution, especially when the system is heavily loaded. With multiple tasks running in the system, the machine with a large memory has more potential power to process tasks and has more opportunities to empty physical memory for other tasks so as to decrease the frequency of exchanging pages with disk. Based on this characteristic, the memory size is used to indicate the memory capacity in our system environment.

The total physical memory is used as the host memory capacity. This number is in units of Megabytes. The six host memory capacities are listed in table 2.6.

2.3.3 Calibration of I/O capacity

Our tests are performed with various file sizes which range from 100M to 2.1G. Five groups of data are obtained with the file sizes as: 100M, 300M, 700M, 1.4G, 2.1G. The response times and bandwidths of *cp* operation on each host are listed in tables 2.3 and 2.4.

Table 2.3 Response time for *CP* files

Hosts	Response Time (s) 100M	Response Time (s) 300M	Response Time (s) 700M	Response Time (s) 1.4G	Response Time (s) 2.1G
host1	50	140	261	484	773
host2	42	120	251	542	747
host3	45	130	313	484	801
host4	46	126	250	467	747
host5	30	99	197	383	603
host6	48	147	349	673	925

Table 2.4 Bandwidth for *CP* files

Hosts	Bandwidth (M/s) 100M	Bandwidth (M/s) 300M	Bandwidth (M/s) 700M	Bandwidth (M/s) 1.4G	Bandwidth (M/s) 2.1G
host1	2.00	2.14	2.68	2.89	2.72
host2	2.38	2.77	2.79	2.58	2.81
host3	2.22	2.31	2.24	2.89	2.62
host4	2.17	2.38	2.80	2.99	2.81
host5	3.33	3.03	3.55	3.66	3.48
host6	2.08	2.04	2.00	2.08	2.27

Figure 2.1 presents the I/O performance for each host with various file sizes. The I/O performance is represented by the data transfer rate (Bandwidth, Megabyte per second). From this figure, we can see that host1, host2, host3 and host4 have similar performance curves and that they exhibit almost the same performance

results when the test files become larger. Host5 exhibits better I/O performance than the other five machines, and host6 has the lowest I/O performance among the six machines.

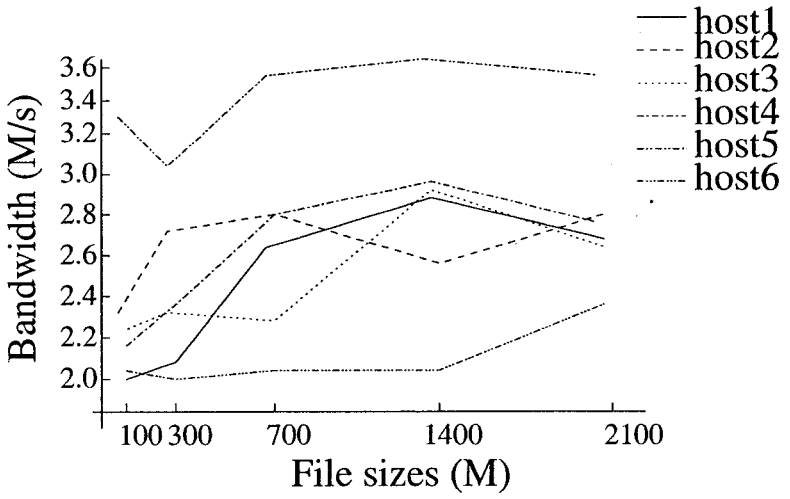


Figure 2.1 I/O performance with variable file sizes.

Based on the Fig. 2.1, we can see that once the test file size becomes greater than 700M, the I/O bandwidth becomes relatively stable. The average of the values obtained at 700M, 1400M and 2100M are used to represent their I/O power. These values are listed in Table 2.5. Various factors may influence the performance while we take our tests. For instance, many users have access to the machines, which may place each host in a different execution state. Considering these environment factors, we assume the I/O systems of the four hosts (host1, host2, host3 and host4) to have equivalent performance.

A small bandwidth value means that the host has a small I/O capacity, and a big bandwidth value means that the host has a large I/O capacity. In our experiment environment, the number of buffer reads/writes between buffer and disk(s) is set as the I/O load index. When the host is very busy, this number may reach up to 4000 or 5000. In order to compare this I/O number with the I/O system capacity

Table 2.5 Results for *CP* file

Hosts	Bandwidth (M/s)
host1	2.76
host2	2.76
host3	2.76
host4	2.76
host5	3.56
host6	2.12

conveniently, we map the values in Table 2.5 to a 0 to 5000 range by multiplying these values by 1000. According to the I/O load formula expressed above, this enlargement does not influence the comparisons of the I/O load values. The purpose of this enlargement for the bandwidths is only to make the I/O load values more readable.

The I/O capacities for the six hosts are listed in table 2.6.

Table 2.6 System capacities

Hosts	CPU Capacity	memory Capacity	I/O Capacity
host1	89	512	2760
host2	89	256	2760
host3	80	256	2760
host4	89	256	2760
host5	229	512	3560
host6	80	256	2120

CHAPTER 3

LOAD BALANCING FOR THE VADOR FRAMEWORK

Based on requirements stemming from MDO applications, the VADOR system aims to integrate various kinds of software packages from different engineering departments. These software packages were implemented in different programming languages and based on different platforms. They have been used for a long time and incorporate large amounts of professional knowledge. A key feature of the system must thus be a seamless integration of the packages in the system providing users with a unique interface that allows them to run simulations across different operating systems. This requirement has led to the choice of a Java-based system to integrate applications and data management in a unified framework.

The VADOR system is implemented in a way that aims to achieve this integration objective within a unified framework. The load balancing subsystem of VADOR tightly integrates into the framework to help utilize resources effectively across a network.

3.1 VADOR Task classification

In Dynamic Load Balancing algorithms, tasks are classified into CPU-bound, Memory-bound, and I/O-bound classes. User can define this attribute when a task is created in the system. In this case, the user's professional knowledge and empirical experience plays an important role. Experienced users often know what are the important characteristics of a task and what can be expected in terms of resource consump-

tion. Another approach is to collect task execution data and analyze that data. When a task is repeated many times over using the execution system, its execution information (CPU, memory and I/O utilization, elapsed time ...) is collected and saved in the database. Analyzing this information, statistical attributes can be extracted for a given task, and this information can be used to assess an expected task resource consumption for the load balancing algorithm.

For a task, the following information is collected:

- CPU-time: how much cpu time the task used during its execution.
- Memory-occupied: When the task is running steadily, how much resident memory it occupies.
- I/O-number: During the lifetime of the task's execution, how many I/O operations happened.
- Elapsed-time: how long the task exists in the system. This time begins from the time the task is scheduled until it finishes.

With this information, tasks are classified into different sets: CPU-bound, Memory-bound, and I/O-bound, CPU-Memory-bound, CPU-I/O-bound, CPU-Memory-I/O-bound, Memory-I/O-bound. The classification of Memory-bound and I/O-bound should be defined according to the characteristics of the workload. We refer to the experiment environment used by Xiao *et al.*, 2002, and define memory-bound programs as programs for which working memory is half or more of the main physical memory of a standard reference node in the system. For the I/O-bound programs, if their I/O rates are very high, we can define them as I/O-bound. In a real working environment, the algorithm of our application software is stable and the behavior can be predicted. Users can define I/O-bound tasks according to their

experiences. In a typical computing environment (Xiao *et al.*, 2002), programs with the average I/O rate of 2 ($IO_s/_{second}$) or over are considered as I/O-bound.

In our algorithm implementation, we define the resources (CPU, memory and I/O system) as having different priorities ($memory > CPU > I/O$). We assume that the memory has the most important influence on the system performance, followed by the CPU utilization, and finally I/O activity. If a program depends on more than one resource, the program will be scheduled considering the resource with the highest priority. For example, to schedule a CPU-bound and memory-bound task, the memory load state is treated as the prominent resource, which means that the host load is evaluated using the memory load index. In order to simplify system implementation, task sets are grouped as:

- *memory-bound*: memory-bound, memory-CPU-bound, memory-I/O-bound, memory-CPU-I/O-bound;
- *CPU-bound*: CPU-bound, CPU-I/O-bound;
- *I/O-bound*: I/O-bound.

The tasks which are neither CPU-bound, nor memory-bound, nor I/O-bound are put into the *CPU-bound* group when they are scheduled by the load balancing system.

For the purpose of our algorithms, we define two important task attributes, which will be used by all dynamic load balancing algorithms. These attributes are used to group tasks into different sets, with tasks in a given group sharing similar characteristics. The attributes are :

- memory consumption;

- task category (CPU-bound, memory-bound and I/O-bound).

Memory consumption indicates how much memory a task generally occupied. When a process runs stably, this value can be recuperated through a system command.

CPU, memory or I/O -bound describes the type of resource that a task will consume most. For some tasks such as engineering or scientific computing, CPU consumption plays an important role. In other cases where file handling becomes important (such as Database transactions and File transmission), care should be given to the I/O system.

3.2 VADOR Load balancing algorithms

3.2.1 Static load balancing algorithm

For the Random algorithm, the task information and system load information are not considered. When a task comes, a host is selected randomly within a host group. The algorithm works as follows:

1. A task comes, applying for a destination host;
2. A random number is created to select the host;
3. The host associated to the random number is chosen for the task;
4. The task is sent to the chosen host.

In the system, a task can be running in a group of hosts or any host across the network. This algorithm is guaranteed to select all hosts evenly in the system.

Although the Random algorithm can choose hosts evenly, it does not consider the difference between hosts, such as current burden, host power and task characteristics. If a host has a light burden, it should be more readily chosen than others. On the other hand, hosts under heavy burden should be avoided in new tasks assignment.

3.2.2 Dynamic load balancing algorithms

This section presents the four dynamic load balancing algorithms.

3.2.2.1 Load-based dynamic shortest load balancing algorithm

The load-based dynamic shortest load balancing algorithm (Simplified as *Shortest*) only uses the current host load to determine host assignment. The following load indexes are thus used by the algorithm:

The CPU load:

$$L_{cpu} = I \tag{3.1}$$

where I is the percentage of idle CPU-time.

The memory load:

$$L_{mem} = M_{free} \tag{3.2}$$

where M_{free} is the amount of free memory.

The I/O load:

$$L_{io} = \frac{1}{N_{io}} \tag{3.3}$$

where N_{io} is the number of buffer reads/writes between system buffer and disks.

Algorithm:

For a given task, the task attributes are used to select the appropriate load index. The CPU idle percentage value, memory free value, and system buffer activities value are used as host load indexes to estimate the candidate hosts. The host capacities, on the other hand, are not considered in this algorithm. The algorithm works as follows:

1. get the task attributes, expected memory and resource bound information;
2. select the hosts which satisfy the tasks memory requirement;
3. if the task is CPU-bound

   select the L_{cpu} as the load index;
4. if the task is Memory-bound

   select the L_{mem} as the load index;
5. if the task is I/O-bound

   select the L_{io} as the load index;
6. calculate the host load values according to the selected load index for the host sets;
7. select the most lightly loaded host (with the smallest load value).

The chosen host is then returned to run the task.

3.2.2.2 Load-capacity-based dynamic load balancing algorithm

In this algorithm, task attribute informations are used, but the system load includes current states and host capacity. The CPU idle percentage value, host CPU capacity, free memory value, memory capacity, system buffer activities and host I/O capacity are considered in the algorithm. The load indexes are now determined as follows:

The CPU load:

$$L_{cpu} = I \times C_{cpu} \tag{3.4}$$

where I is the percentage of idle CPU-time, and C_{cpu} is the capacity of CPU.

The memory load:

$$L_{mem} = M_{free} \times C_{mem} \tag{3.5}$$

where M_{free} is the amount of free memory and C_{mem} is the memory capacity.

The I/O load:

$$L_{io} = \frac{C_{io}}{N_{io}} \tag{3.6}$$

where N_{io} is the number of buffer reads/writes between system buffer and disks

and C_{io} is the capacity of I/O system.

The definitions and calibrations of the C_{cpu},C_{mem} and C_{io} are detailed in Chapter 2.

This algorithm uses the same host selection procedure as Load-based dynamic shortest load balancing algorithm except for different definitions of load indexes.

3.2.2.3 Load-CPU-based dynamic load balancing algorithm

In this algorithm, only the CPU load is used to express the load index. The current memory load and I/O load are not considered in the assignment of tasks. The load index is defined as:

$$L_{cpu} = I \tag{3.7}$$

where I is the percentage idle of CPU-time. The algorithm is expressed as following:

1. get the task information: expected memory;
2. select the hosts which satisfy the tasks memory requirement;
3. calculate the host load values according to the load index for the host sets;
4. select the most lightly loaded host (with the smallest load value).

3.2.2.4 Load-based threshold dynamic load balancing algorithm

This algorithm selects a host with its host load above a threshold. The task's properties are used while evaluating the host load. If the task is CPU-bound, the current CPU state is treated as the load index. If the task is memory-bound, the current free memory is used as the load index. If the task is I/O-bound, the system buffer reads/writes number with the disks becomes the load index. The host capacities are not used in the load indexes.

The CPU load:

$$L_{cpu} = I \tag{3.8}$$

where I is the percentage idle of CPU-time.

The memory load:

$$L_{mem} = M_{free} \tag{3.9}$$

where M_{free} is the amount of free memory.

The I/O load:

$$L_{io} = \frac{1}{N_{io}} \tag{3.10}$$

where N_{io} is the number of buffer reads/writes between system buffer and disks.

For each resource, a threshold is defined to distinguish the host load state. Suppose

that T_{cpu} is the CPU threshold. If the CPU load is higher than this value, the host is in light load, otherwise, the host is in heavy load. Let T_{mem} be the memory threshold. If the free memory is higher then this value the host is thought as light, otherwise the host is in heavy load. Let T_{io} be the I/O threshold. If the I/O load (system buffer activity number) is above the value, the host is thought as lightly loaded, otherwise the host is heavily loaded.

The algorithm works as follows:

1. get the task attributes, expected memory and resource bound information;
2. select the hosts which satisfy the tasks memory requirement;
3. if the task is CPU-bound

   select the first host with $L_{cpu} > T_{cpu}$;
4. if the task is Memory-bound

   select the first host with $L_{mem} > T_{mem}$;
5. if the task is I/O-bound

   select the first host with $L_{io} > T_{io}$.

3.3 Load balancing system implementation in VADOR

3.3.1 System interaction protocol

Java has strong networking and distributed computing capabilities. Sockets, RMI and Servlets enable programmers to develop distributed applications in Java. The socket-based communications enable applications to view networking as if it were

file I/O. A program can read from a socket or write to a socket as simply as reading from a file of writing to a file. Basic network programming with sockets is flexible and sufficient for communication between programs. It requires all involved parties to communicate via application-level protocols. Sockets involve network communications at a fairly low level that require detailed handling of the socket setup protocols.

In the VADOR system, sockets are used as the basic communication mechanism. These communications occur between the GUIs and the Executive server, the Executive server and the Librarian server, the Executive server and the Wrapper servers.

For flexibility in system implementation, the Proxy pattern (Gamma *et al.*, 1995) is introduced into the communication between different system components. A proxy usually has the same methods as the object it represents, and it passes on the method calls from the Proxy to the actual object.

3.3.2 The VADOR framework

The VADOR system provides users with a unique GUI to create, execute, and control engineering analysis tasks. Once a task is created and sent to run, a central management module is responsible for assigning a host in the network to run the task. The load balancing system is used in the central management module helping to select a lightly loaded host. During the system execution, host load informations are collected across the network and gathered for decision-making purposes in the load balancing system.

The VADOR framework includes a GUI, an Executive Server, Wrapper Servers and a Librarian server. The respective functions of each server are expressed as

follows and their integration is outlined in figure 3.1.

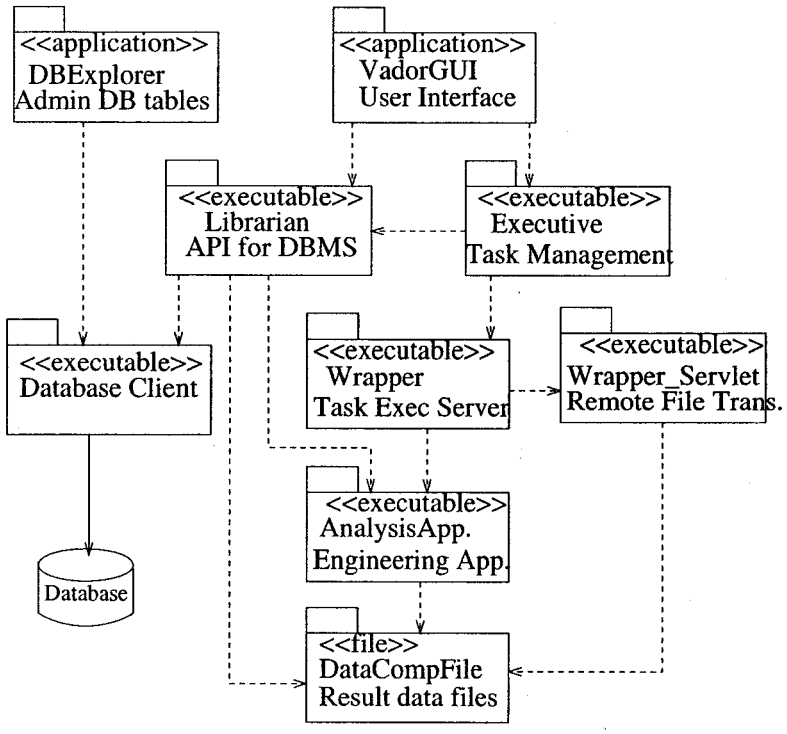


Figure 3.1 Vador architecture

VADOR GUIs: applications through which most users interact with the system. Users can create tasks, control tasks and monitor the system status. Through GUIs, users send executable tasks into the Executive server that allocates the execution resources.

VADOR Librarian: server that allows to decouple the system API from the underlying database server and file servers capabilities. The system data (created tasks, execution information, statistics data etc.) are stored in a database through this server.

VADOR Executive: server which manages the execution of various tasks. When Users launch tasks through the GUI, the Executive server assigns them to suitable Wrapper Servers (CPU servers). It receives the response message from Wrapper

servers and sends them back to GUIs, and saves results into the database.

VADOR Wrapper: servers which run wrapped analysis packages. The VADOR tasks are sent to Wrapper servers for execution. They also collect system status and send them to the Executive server.

3.3.3 Load balancing in VADOR

Through GUIs, users launch various tasks that need to be executed by the system. These tasks are sent to the Executive server that schedules system resources. When the Executive server receives a task, it must select a Wrapper server (cpu Server) as the destination. As a result, Wrapper servers load varies dynamically. Load balancing issues arise with the assignment of the tasks. As mentioned in chapter 1, load balancing algorithms can assign work loads evenly across the distributed system. While there are many kinds of algorithms researched in distributed system literature, in our project, a selected set of Dynamic Load Balancing algorithms has been chosen and their performance has been compared to the Random algorithm in the specific VADOR application field.

In the random algorithm, the Executive server receives incoming tasks and assigns them to the hosts (Wrapper servers) randomly across the network within an a priori known group of servers that can run the task. Every host has the same opportunity to run a job. When a task is launched, it becomes automatically associated with a group of hosts based on a priori knowledge about the task. The algorithm module creates a random number to select a host within the hosts group. The incoming task is then sent to the chosen host for execution.

The Dynamic Load Balancing algorithms use the entire system load to make an estimation for each host in a group of hosts. Based on these load estimations, the

host with the best value is chosen.

3.3.4 Load balancing subsystem architecture

As illustrated in Figure 3.2, the load balancing subsystem includes two parts: the global collector and the local collector.

global collector: embedded in the central host (Executive Server). It is responsible for collecting host load across the network and assigning incoming tasks to hosts using one of the load balancing algorithm. The global collector gathers system load periodically and manages the utilization of hosts. The load balancing algorithm runs within this module and supplies proper hosts to incoming tasks. The function of checking local collectors activities is implemented here. If some local collectors crash during system execution, a restart mechanism is a part of the module's responsibilities.

local collector: resides on each host to collect host information. It gathers host information periodically and makes an estimation to decide whether the host state has experienced a significant change since the previous information transmission. If the new host state values need to be sent to the global collector, a communication is established between this local collector and the global collector. Through the information exchange channel, host load values are transmitted to the global collector. After that, the local collector disconnects from the global collector, sleeps for a period and enters into the next information gathering loop.

Following is a brief description of the main components and classes that constitute each module in the load collection subsystem.

host load: This class encapsulates the data of individual host load. These values

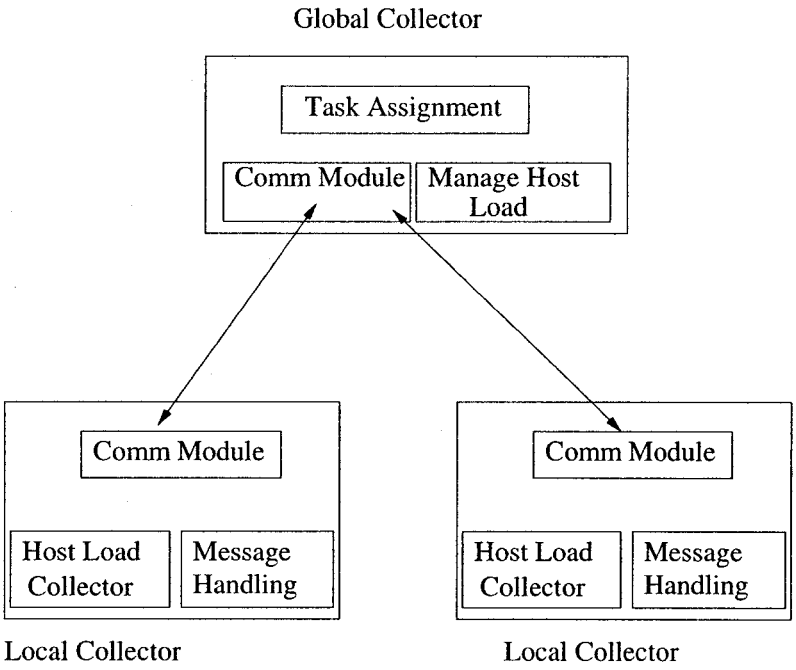


Figure 3.2 Load balancing architecture

are collected in local collectors and passed to the global collector. An object of this class is a snapshot of the current host resource utilization.

Dynamic load balancing algorithms use host load to estimate system burden. Local collectors gather host status periodically and sends them into the global collector. In our algorithms, the CPU, memory and IO information are considered while tasks are scheduled.

The host load includes: the percentage idle of CPU time, free memory, total memory, and I/O numbers.

3.3.5 Global collector

This subsystem includes three modules as illustrated in figure 3.3: the communication module, the host load management module and the task assignment module.

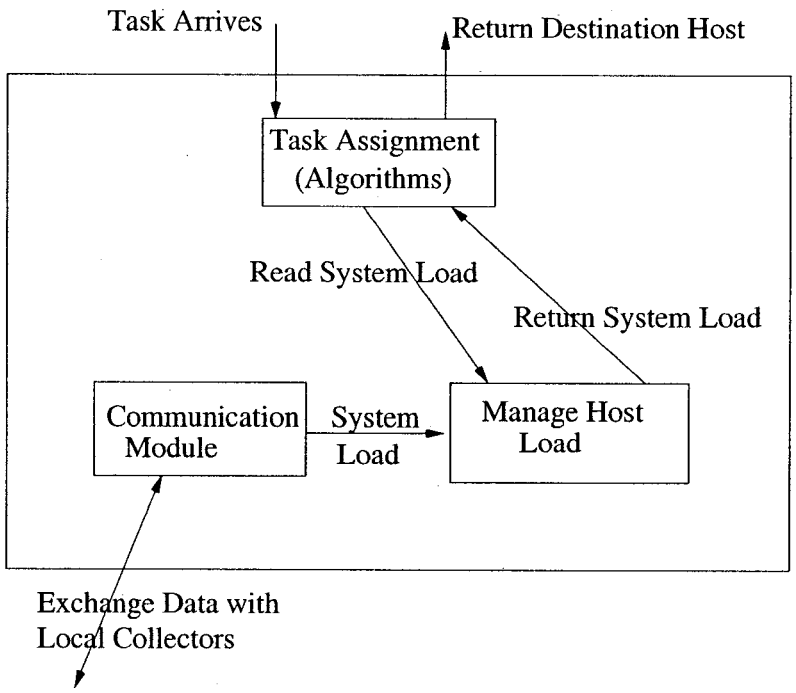


Figure 3.3 Global Collector

communication module: This module is responsible for communicating between the global collector and local collectors. It receives the coming host loads and stores them in the load management module. This module also controls the communication for checking state of local collectors. It builds channels with local collectors and sends requests to local hosts periodically to test whether they are still alive. The local collectors use this communication channel to send their responses to the host load management module.

host load management module: It receives the system loads from local hosts across the network and stores them in its internal data structures. This information is supplied to the task assignment module for decision-making purposes. If the system load monitor is activated, these host load informations can be displayed and dynamically updated when new host information arrives from local collectors. This allows users to monitor the system dynamic status in real time.

In order to guarantee that local collectors are working, the global collector sends commands to local collectors to check their status, as illustrated in figure 3.4. If a local host does not respond for several time periods, it is assumed that the local host is dead. To handle a dead local collector, the load management module locks the host's load value to prevent its use in the future. This lock is not released until the local collector is restarted and the new host load values arrive. The dead local collectors can be restarted manually by users.

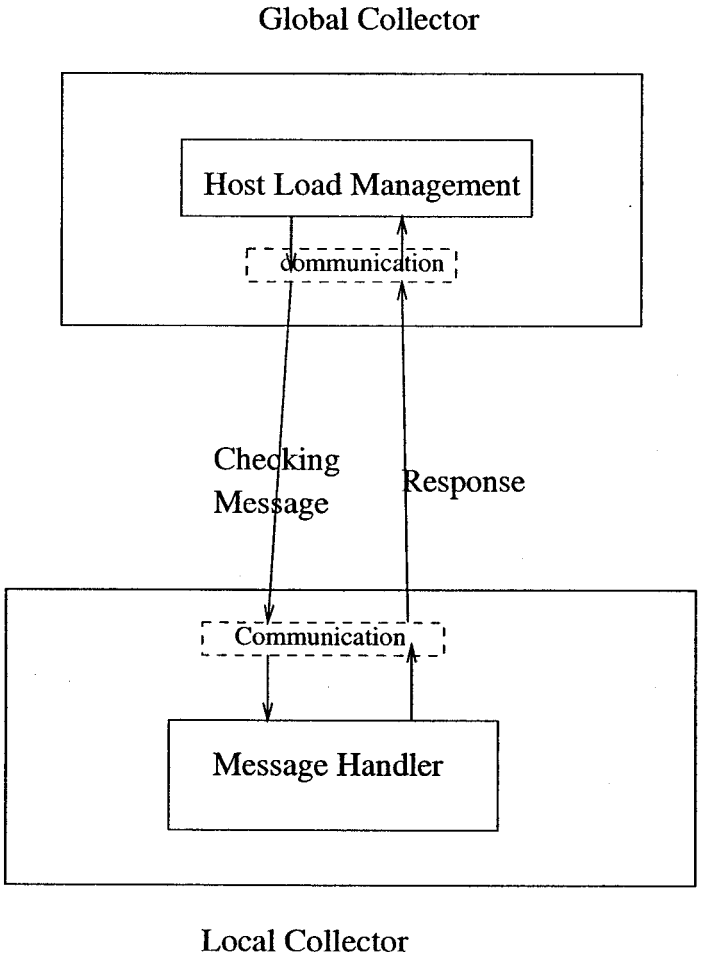


Figure 3.4 Local Collectors State Checking

When local collectors crash, remote restart are needed to reactivate them. On Unix systems, a *rsh* command is used to launch the program on the remote machine.

task assignment module: This module is used to choose a destination host from the system. When a task arrives, this module first retrieves the task information from the database, then it gets the system load from the host load management module. Using the system load information, the load balancing algorithm is used to select a host using one of the implemented algorithms.

3.3.6 Local collector

This subsystem includes three modules: the communication module, the message handling module and the load collection module.

communication module: This module communicates with the global collector. On the local host, when load data is collected, it is sent to this module. This module establishes a connection with the global collector communication module, and sends the information to its peer. Once the transmission is finished, the connection is closed. During the data transmission, some exceptions can be handled. If the network is unavailable, a connection can not be built, the module gives up the data transmission and the new host load is thrown away. If an IO exception occurs, the host load is also discarded.

message handling module: This module receives and deals with the messages from the global collector. When the global collector sends a status checking message, a confirmation response is created and returned to the global collector.

load collection module: This module is responsible for host load collection. At every time interval P, the host load is collected. Suppose that time t is used as the collection period, and that t is broken into n time slices. For each t/n interval, the CPU Idle, free memory, and buffer activities are gathered. Once time t has elapsed, the n sets of values are averaged to represent the host load metrics.

In this module, the newly computed host load metrics are compared with the previously sent values. If there are significant differences, the new host metrics are sent to the communication module, and transfered to the global collector. If the new host load metrics have not changed significantly over the last period, they are thrown away and do not need to be sent to the global collector.

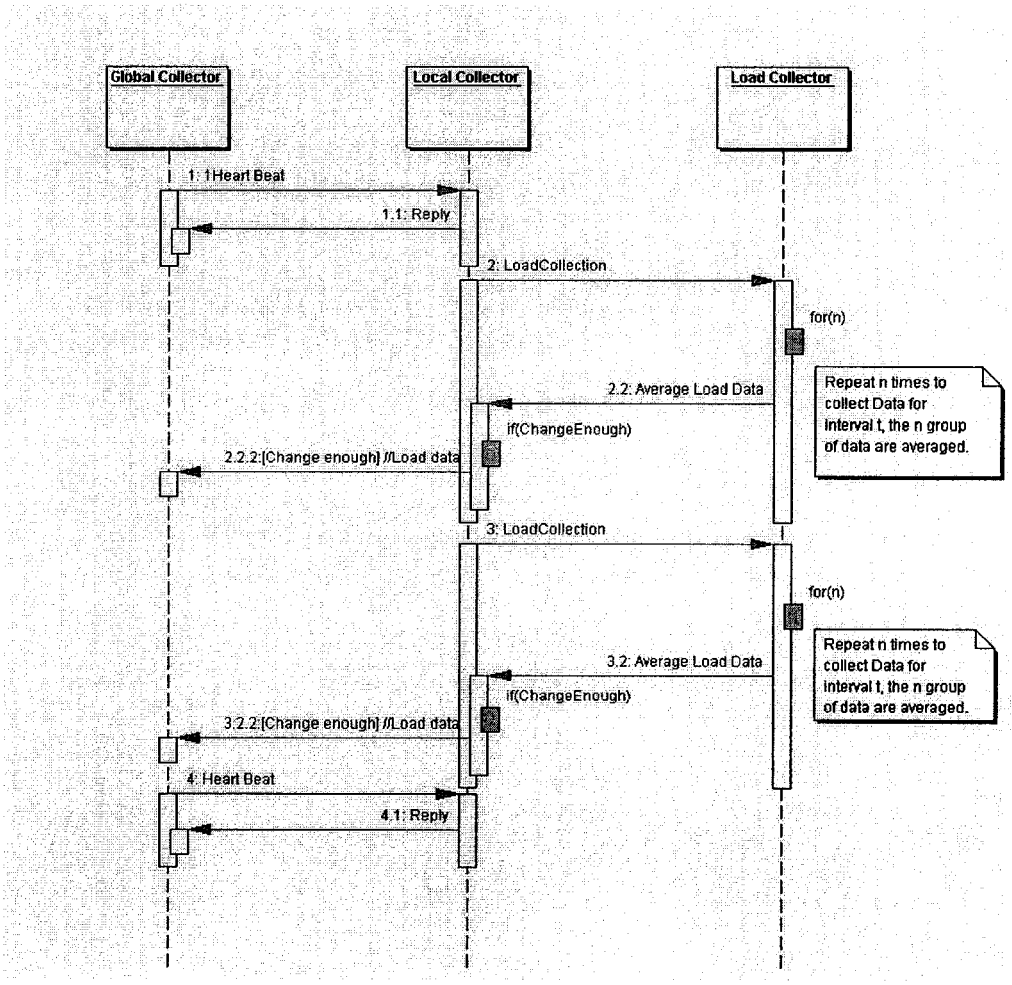


Figure 3.5 Messages between global collector and local collectors

Figure 3.5 presents the messages passing between the global collector and local collectors. In order to simplify the figure, we suppose that two load collections are done when another checking message is sent.

The following two Unix commands are used to collect host information.

sar — system activity reporter.

This command collects the CPU utilization, memory information and system buffer activity for a period. The continuous collection can be performed for n time intervals. The averaged data can be extracted for the n sample data.

top — displays and updates information about the top cpu processes.

Top displays the most CPU time consuming processes on the system and periodically updates this information.

Top can be used in batch mode and in that case only outputs numbered displays. A display is considered to be one update of the screen for the system information and processes status. This command displays the operating system, load average, cpu information (such as idle, usr, system), and memory (Max, free, ...).

On IRIX64 and Linux systems, the CPU utilization, buffer activity and memory information are collected using the *sar -ubr* command. From the data of CPU utilization, the idle percentage of CPU time is extracted. From the buffer activity data, the transfer rate of basic blocks between system buffers and disk is extracted. From the memory information, we obtain the free memory data. Once these informations are collected, they are sent to standard output with special strings to mark them as follows:

- vador_cpu_idle: XXXXX
- vador_buf_num: XXXXX
- vador_mem_free: XXXXX

This procedure is accomplished by a *shell* script. The script is invoked from a JAVA program and the outputs are captured, analyzed and sent to the *global collector.*

On IRIX64 systems, the *top* command is used to collect the total host memory when the *local collectors* are launched. The *top* command is executed in a *shell* script and in the output of the *top* command, the total physical memory field is extracted. In this script, the data is formatted to standard output as:

vador_mem_total: XXXXX

On Linux system, the *sar -r* command is used to collected the total host memory when the *local collectors* are launched. The *shell* script analyzes the output of the *sar -r* command and extracts the total memory data to output in the same format above.

A JAVA program is used to encapsulate these scripts and capture the standard output. The total memory data is thus read by *local collectors* and sent to the *global collector.*

3.4 Analysis of task information

In the dynamic load balancing algorithms, task attributes are used to help host assignment. These attributes can be obtained from the tasks execution information gathered during the tasks execution. The tasks execution information includes: user time used by the task, system time used by the task, memory occupied in stable operating mode, I/O information and elapsed time of the task (from entrance into the system, until the execution finishes). This information can also be used to analyze system performance.

In the VADOR system, the task information collection module is attached to the Wrapper servers. It is automatically invoked when the Wrapper server runs a task. When the task ends, the execution information is collected and sent to the Executive server to be saved in the database. Category analysis can be done based on these execution informations stored in the database.

The architecture of the task information collection module is illustrated in figure 3.6.

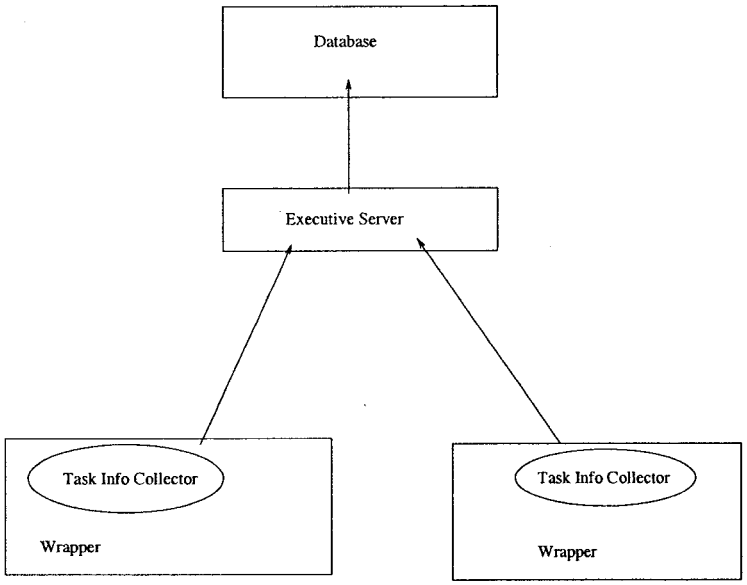


Figure 3.6 Task Collection Architecture

The task information collection module works in conjunction with the Wrapper server, when a process is created, the process *handle* is passed into the task information collection module. This module waits for the task to finish and collects the process information. It works as illustrated in figure 3.7.

Tasks associated with same program(s) or software package(s) are considered to belong to the same type. Tasks of the same type are grouped, their average occupied memory, average cpu time used for execution, average elapsed time in system, and

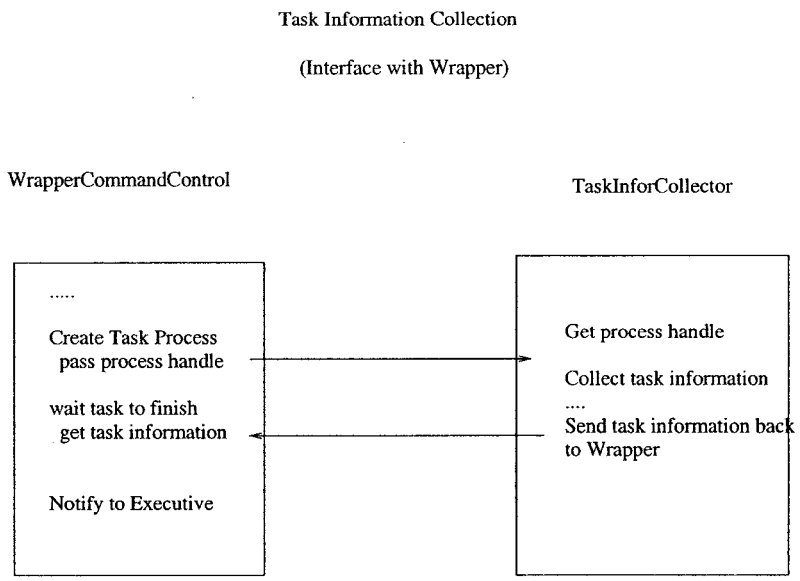


Figure 3.7 Task Collection

average I/O operation are analyzed. With this information, tasks can be classified into CPU-bound, memory-bound, or I/O-bound.

Following is a short description of the Unix commands used in the task information collection module:

time — time a command

This command is executed, after it exits, time prints resource usage statistics to standard error. The following information can be collected:

- Elapsed time the command spent running;
- CPU time spent in execution of the command;
- CPU time spent executing system code by the command;
- Maximum resident memory;
- The number of disk input and output operation.

An alternative method, which has also been tested with comparable results, uses the getrusage or wait3 function in C language to collect information.

getrusage — get information about resource utilization.

wait3 — wait for child process to stop or terminate.

The two functions return information describing the resources utilized by the current process, or all its terminated child processes. The following information can be collected:

- user time used in execution of process or all its child processes;
- system time used in execution of process or all its child processes;
- The maximum resident memory utilized;
- The number of times the file system had to perform input and output.

When a task arrives to execute, the execution command (a script with parameters) is constructed by the Wrapper server. In order to obtain the execution informations of the command, a new script is created with the execution command as the input parameter. In the new script, the execution command is the parameter of the command *time*, and the standard outputs and standard errors are captured and analyzed. The task execution information is extracted from this information and formatted to be read by the task information collection module. For portability across different Unix platforms, task execution informations is formatted to a unique style in this new script.

CHAPTER 4

COMPARISON OF LOAD BALANCING ALGORITHMS IN VADOR

4.1 Introduction

In this chapter, a set of test programs are executed on a set of hosts interconnected through a communication network. The distributed load balancing algorithms described in Chapter 3 are used to determine which host will actually run which task. The behavior of the different algorithms is analyzed under various system loads.

In the load balancing subsystem, Random and Dynamic load balancing algorithms are implemented. All these algorithms can be run through the VADOR load balancing system and switched seamlessly. All tasks are created across the network and sent to the global collector which is running in the central management node. These tasks are scheduled in the global collector and a host is selected to run them.

The VADOR system is launched in the test environment. In order to spread the system load more evenly, the Executive and Librarian servers are installed on separate hosts. The database is installed on a server which does not belong to our test environment.

Local collectors are set up on each host (together with the Wrapper server). The global collector is configured in the Executive server.

A simple VADOR client is used instead of the VADOR GUI to launch tasks from a client host and send them to the Executive server. A batch of tasks can be

scheduled and the interval time between two tasks can be set here. The simple VADOR client switches different algorithms to run in the global collector and outputs task execution informations once the batch of tasks are finished.

4.2 Workloads

Tasks are classified into three categories : CPU-bound, Memory-bound and I/O-bound. The proportion of each type of tasks composing the workload can influence algorithmic performance. In our test workloads, the percentage of tasks belonging to each of the three classes are defined as: r_{cpu} (percentage of CPU-bound tasks), r_{mem} (memory-bound tasks) and r_{io} (I/O-bound tasks occupy). With :

$$r_{cpu} + r_{mem} + r_{io} = 100\%$$

We have selected seven scientific and engineering programs which are representative of CPU-bound, memory-bound and I/O-bound tasks. The following are brief descriptions for each of them.

- Queens: The Queens program rates the time a computer takes to find all ways that n queens can be placed on a n X n "chess" board such that no queen can capture another. This program is a CPU-bound task (Netlib, 2002);
- Pi: This program computes the *pi* number to arbitrary precision. We control the program's execution by configuring the expected length of *pi*. This program is a CPU-bound task;
- Optim-Nurbs(O-N): This program optimize wing profiles represented as NURBS curves . This is a standard benchmark application used in CERCA which

includes three subtasks. This is a CPU-bound task;

- Matrix multiplication(m-m): This is a standard matrix multiplication program and it is a memory-bound task (Xiao *et al.*, 2002);
- Hanoi: An integer program that solves the Towers of Hanoi puzzle using recursive function calls. We modified this program to output large amounts of data and become an I/O-bound task;
- SumTwist: This is a small program which calculate the curves. It classifies as a CPU-bound task;
- Loopsh: This is our own test program which outputs a string message for n repeats. It is an I/O-bound task.

We have measured the execution of these programs manually and monitored their execution time (user CPU, system CPU and elapsed time), resident memory and I/O activities (reads and writes numbers). Our computing environment is the host3 in Table 2.2, a *SGI INDIGO*^{2}.

Table 4.1 presents the experimental results of the above seven programs on *host3* in table 2.2. The *CPU* figure expresses the CPU time used by each program, which is the sum of *user* time and *system* time, the *elapsed* figure refers to the lifetime of the program, the *memory* figure expresses the resident memory while the program is running and the *I/Os* figure refers to the sum of read and write numbers during the program life periods.

The Optim-Nurbs program includes three sub-programs (O-N-1, O-N-2 and O-N-3). The execution informations of the three sub-programs are listed separately in table 4.1.

Table 4.1 Execution Performance of Application Programs

Programs	CPU(s)	elapsed(s)	memory(MB)	I/Os
Queens	24.91	34.67	1.14	268
Pi	7.86	10.65	0.91	2
O-N-1	0.30	0.38	0.92	1
O-N-2	0.18	0.29	4.11	2
O-N-3	0.42	0.74	4.77	1
m-m	227.77	364.53	123	3
Hanoi	21.04	95.75	0.88	6973
Loopsh	4.32	8.67	0.53	200
sumTwist	0.03	0.12	0.03	3

4.3 Experimental task scheduling

To compare algorithmic performance under various system loads, different task creating frequencies have been used in the experiments. We use the parameter f as the frequency of incoming tasks, with $f = \frac{1}{p}$, where p is the time interval between successive incoming tasks, or period of arrival.

This value f aims to indicate the entire system load. When f becomes large, more tasks are created in a fixed duration, the system load increases. On the contrary, for small values of frequency, small number of tasks are created in a fixed duration, and the system becomes lightly loaded.

Several schedules are defined for different system loads. The schedules are classified into light, middle, and heavy. The *sleep* command is used to simulate the intervals between the origination of tasks. In our experiments, we create schedules with different time intervals, such as: 20s, 30s, 40s, 50s, 60s, 70s, etc. Tasks are defined and arranged randomly in a list. The interval time between tasks can be set to the above interval values to produce different workloads. Once the interval time is set

in the schedule list, this schedule can be launched and the system will automatically send a task to the Executive server and wait for the *interval time* to elapse. When the wait for the *interval time* ends, the next task is sent to the Executive server. This process loops until all the scheduled tasks have been sent to be executed.

In order to reduce the impact of external factors, most of our experiments were carried out at night and during weekends. Each experiment was repeated 5-7 times and the average results were computed. After each execution of the task schedule, the performance metric was computed based on the execution results of the tasks in this schedule.

4.4 Parameter setting of local collectors

As discussed in Chapter 3, local collectors are controlled through three parameters:

P — the period of one collection;

T — the collecting time for one sub-collection;

C — the count number for the sub-collections.

The local collector starts a collection operation every P seconds. During the period P, only $T \times C$ seconds are used to collect load information. Each set of load values are collected for a period T. And this process is repeated C times without interrupts. The result state is the average of the C sets of load values. Once the result state is obtained, the local collector sends it to the global collector if necessary and sleeps to wait for the next collection period.

The definition of the three parameters should be closely related to the system and application workload characteristics. With low arrival rate of tasks, P can be longer to reduce the host burden. With high arrival tasks rate, P should be shorter to reflect immediately the state changes. T represents the accepted time for which old information can be used. If T is too long, old and out-of-date information is used in the decision-making. With shorter T, the estimation of host load is easily influenced by sudden peak values.

4.5 Performance metrics

The metrics used to measure the system performance are the mean response time, and the mean slowdown. The response time is the amount of time needed by a task entering the executing host to finish. The slowdown is the ratio between the response time and the CPU time used by the task (please see § 1.7). The average response time is defined as the average task response time over all the tasks and the mean slowdown is the average slowdown of all the tasks.

4.6 Network traffic

In the Dynamic load balancing algorithms, no consideration is given to the network traffic. The network traffic may influence the system performance when the file transfers are needed between different hosts. Based on the following reasons, we ignore the network traffic in our algorithm analysis and implementation.

- In the VADOR system, only simple messages are sent between the *Executive Server* and the *Wrapper Servers*. These messages are transfered when tasks

are sent to *Wrapper Servers* and their executions finishes. These messages include little data and can not cause significant network traffic;

- Most of our programs perform long running tasks compared with the network communications, and the scientific calculations play an important role in programs executions. The network traffic only occupies a very little part in the tasks lifetime;
- There are no communications among tasks while they are running;
- In our test workloads (§ 4.2), there are not many big files that need to be transfered frequently between hosts. And the output files are created on local disks.

4.7 Tests on a homogeneous environment

4.7.1 System environment

Mostly for algorithmic validation purposes, and as a simplified test environment, a homogeneous computing environment has been established to perform initial load balancing system tests. This system includes 4 high speed workstations (mono-processor x86) connected through a high speed network. Each workstation has 1G memory and the entire system shares the same disks. A Linux kernel version 2.4.7 was used to control the cluster system.

The following programs were used to constitute the workload.

- Queens;
- Pi;

- Matrix multiplication(m-m);
- Hanoi;
- Loopsh.

Since *SumTwist* is a small program and the test system is made up of high speed workstations, this program was ignored in the workload. Also, because the Linux executable for the *Optim-Nurbs* program was not available, it was also excluded from this workload.

Since this test system used high speed workstations equipped with powerful CPUs, the execution parameters for the above programs were adjusted to prolong the execution time.

For each of the above programs, 20 execution instances (tasks) were created to construct the task schedule. The 100 tasks (20×5) were ordered randomly in the schedule. The control of the schedule execution is described in § 4.3.

The setting of the parameters (§ 4.4) for local collectors were as follows:

$P = 1$ *minute*, $T = 3$ *seconds*, $C = 4$

Because this system is homogeneous, the Shortest-Capacity algorithm became strictly equivalent to the Shortest algorithm. In our experiments, only three algorithms were tested (Random, Shortest and Shortest-CPU).

The cluster system shares the same disks. When our experiments were carried out, the I/O information could not be collected by local collectors individually on the nodes. For the Shortest algorithm, the I/O-bound tasks were assigned randomly.

4.7.2 Results

Table 4.2 presents system performance of the algorithms for a task assignment period of 15 seconds.

Table 4.2 Performance (15s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	38.89s	142.35s	4.18
Shortest	38.81s	160.68s	5.12
Shortest-CPU	38.92s	167.37s	5.29

Table 4.3 presents system performance of the algorithms for a task assignment period of 20 seconds.

Table 4.3 Performance (20s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	38.66s	86.24s	2.48
Shortest	38.35s	77.33s	2.23
Shortest-CPU	38.58s	85.67s	2.46

Table 4.4 presents system performance of the algorithms for a task assignment period of 25 seconds.

Table 4.4 Performance (25s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	38.31s	56.88s	1.55
Shortest	38.44s	60.78s	1.66
Shortest-CPU	38.50s	70.08s	1.92

Table 4.5 presents system performance of the algorithms for a task assignment period of 30 seconds.

Table 4.5 Performance (30s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	38.26s	61.28s	1.71
Shortest	38.16s	51.51s	1.42
Shortest-CPU	38.15s	56.13s	1.51

Table 4.6 presents system performance of the algorithms for a task assignment period of 35 seconds.

Table 4.6 Performance (35s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	38.44s	59.65s	1.65
Shortest	37.74s	49.31s	1.35
Shortest-CPU	38.17s	50.52s	1.38

4.7.3 Analysis

Figures 4.1 and 4.2 present the system performance of the three algorithms (Random, Shortest, Shortest-CPU) in a high speed homogeneous distributed system. Since the system is homogeneous, figures 4.1 and 4.2 look very similar. The three algorithms exhibit almost the same results. When the system is heavily loaded, the Random algorithm yields better system performance than the others. When the system is not heavily loaded, the Shortest algorithm is a little better, then the Shortest-CPU algorithm and at last the Random algorithm. Although load balancing algorithms can improve the system performance for high speed homogeneous computing environments, improvements are rather small among the various algorithms.

When the system is heavily loaded, that is to say, tasks arrive in the system very frequently (a task every 15 seconds), the Random algorithm obtains a better system performance than others. Because *local collectors* update load information at intervals of 60 seconds and each 15 seconds a task is sent to the system, the load informations stored in the *global collector* become too old. When these load informations are used to evaluate hosts, the incorrect assignments are made for incoming tasks. In this situation, more tasks will be sent to a host mistakenly considered lightly loaded, and cause the host to become over loaded. This result has been discussed by Mitzenmacher, 1997.

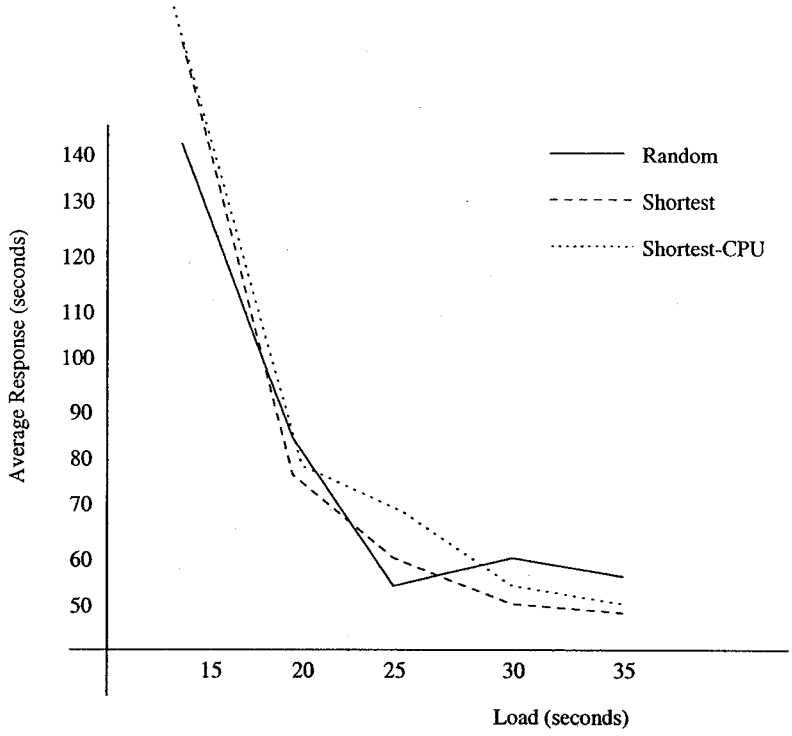


Figure 4.1 Average Response Time.

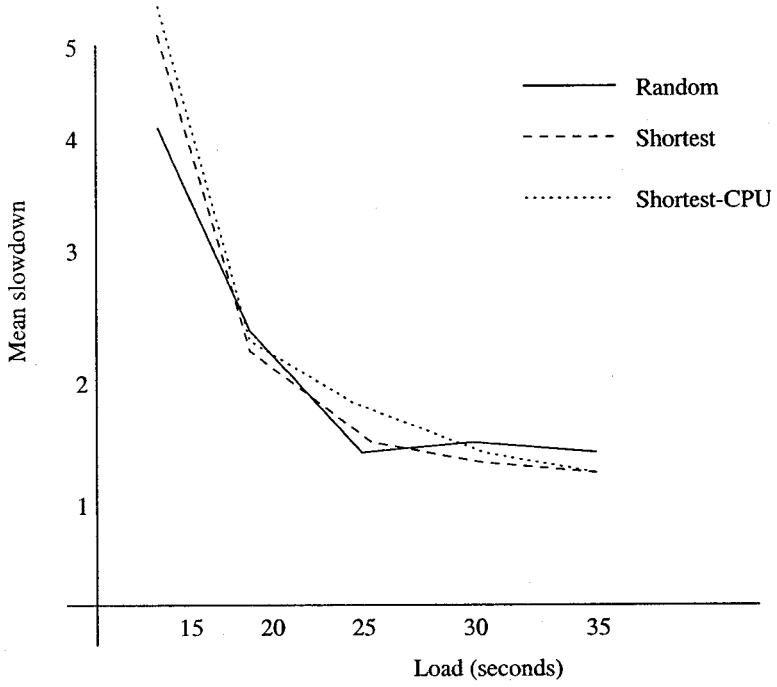


Figure 4.2 Mean Slowdown.

4.8 Heterogeneous test environment

4.8.1 System environment

The bulk of our experiments have been carried out on a distributed system comprising 6 hosts in our local area network all managed through a central server. Wrapper servers were installed on every host.

The configuration of the 6 hosts is described in § 2.

Workloads are composed of the seven programs listed in § 4.2. 64 tasks were created and combined to produce these workloads. The execution order of these tasks was randomly distributed. In the workloads, the percentages of the CPU-bound, memory-bound and I/O-bound tasks were chosen as follows:

$r_{cpu} = 60\%$, $r_{mem} = 20\%$ and $r_{io} = 20\%$.

The setting of the parameters (§ 4.4) for local collectors were as follows:

P = 1 minute, T = 5 seconds, C = 3

4.8.2 Experiments

Five algorithms: Random (§ 3.2.1); Load-based Shortest Dynamic load balancing without the capacities (§ 3.2.2.1); Load-Capacity-based Shortest Dynamic load balancing (§ 3.2.2.2); CPU-based Shortest Dynamic load balancing (§ 3.2.2.3) and Threshold(§ 3.2.2.4) were tested under our experimental setting. We refer to them as: Random, Shortest, Shortest-Capacity, Shortest-CPU and Threshold.

The system behavior with the five algorithms under different system loads was measured.

4.8.2.1 Parameters for the Threshold algorithm

The threshold values are needed for the Threshold algorithm to distinguish the host state (under-loaded, over-loaded). Supposing that the threshold values of the CPU, memory and I/O are represented as T_{cpu}, T_{mem}, and T_{io}. Referring to the discussion in Chapter 1, we set the T_{cpu} at 20. In their paper Xiao *et al.*, 2002 mention that the optimal memory load of a process can be set at 40 percent of its request memory size. According to our workloads, the memory-bound tasks occupy about 130 MB of memory space. In order to avoid insufficient memory state, we chose to set T_{mem} as 50MB (about the $130 \times 40\%$). We have performed some experiments to test the optimal threshold value for the I/O metric.

Table 4.7 presents system performances for various values of the I/O threshold. The workload used in this experiment is a schedule with 40s time interval in task arrival. This workload is an I/O-bound schedule where CPU-bound tasks, memory-bound tasks and I/O-bound tasks have the following percentages:

$r_{cpu} = 40\%$, $r_{mem} = 20\%$ and $r_{io} = 40\%$.

Table 4.7 Performance of different I/O threshold for the Threshold algorithm

Threshold	Mean CPU-used	Mean Response	Mean Slowdown
100	21.21s	107.04s	8.61
200	21.14s	103.50s	10.40
300	21.46s	105.16s	9.47
400	21.63s	110.13s	8.75
500	21.44s	109.48s	9.91
600	21.36s	94.99s	9.04
700	20.31s	99.44s	10.04
800	20.82s	124.64s	11.17
900	20.60s	134.29s	10.91
1000	19.69s	130.75s	16.34

Looking at the mean response time for this group of tasks in our environment, the best values are 94.99 (Threshold value of 600), and 99.44 (Threshold value of 700). For a threshold value above 800, the system mean response time increases. As for mean slowdown, for this group of tasks, the value begins to increase when the threshold value is larger than 700. Below 700, the mean slowdown of tasks does not display large differences. Considering these results both for mean response time and mean slowdown, we selected 700 as the optimal threshold value for I/O numbers.

4.8.2.2 Schedule for light system load

Table 4.8, 4.9, and 4.10 present the performance metrics of the five algorithms for the system in light load state. In table 4.8, the task schedule interval is 70 seconds, in table 4.9, it is 60 seconds and in table 4.10, it is 50 seconds.

Table 4.8 Performance in light load state (70s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	24.58s	46.86s	2.57
Shortest	15.29s	30.19s	2.62
Shortest-Capacity	16.07s	30.39s	3.42
Shortest-CPU	29.35s	49.72s	2.46
Threshold	12.69s	28.59s	4.28

Table 4.9 Performance in light load state (60s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	26.70s	53.10s	2.93
Shortest	15.67s	29.15s	2.13
Shortest-Capacity	15.83s	31.92s	3.44
Shortest-CPU	29.11s	49.30s	2.51
Threshold	13.40s	29.15s	3.95

4.8.2.3 Schedule for middle system load

Table 4.11 and 4.12 present the performance metrics of the five algorithms for the system in middle load state. In table 4.11, the task schedule interval is 40 seconds and in table 4.12, the task schedule interval is 30 seconds.

Table 4.10 Performance in light load state (50s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	27.40s	57.75s	2.89
Shortest	17.46s	35.31s	2.81
Shortest-Capacity	16.15s	33.53s	4.04
Shortest-CPU	26.56s	44.84s	2.34
Threshold	13.14s	33.42s	4.90

Table 4.11 Performance in middle load state (40s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	26.35s	58.09s	4.61
Shortest	15.91s	34.22s	4.12
Shortest-Capacity	15.23s	33.97s	4.29
Shortest-CPU	19.91s	42.22s	4.50
Threshold	12.24s	37.43s	6.17

4.8.2.4 Schedule for heavy system load

Table 4.13 presents the performance metrics of the five algorithms for the system in heavy load state. the task schedule interval is 20 seconds.

Table 4.12 Performance in middle load state (30s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	23.69s	59.32s	4.70
Shortest	13.89s	38.94s	5.61
Shortest-Capacity	14.03s	40.22s	6.04
Shortest-CPU	27.57s	71.06s	5.36
Threshold	13.12s	50.51s	8.91

Table 4.13 Performance in heavy load state (20s)

Algorithm	Mean CPU-used	Mean Response	Mean Slowdown
Random	26.35s	99.85s	7.71
Shortest	15.26s	70.13s	8.51
Shortest-Capacity	14.19s	89.09s	13.09
Shortest-CPU	28.12s	87.31s	6.53
Threshold	13.53s	102.22s	20.68

4.8.2.5 Algorithms success ratio

In our experiments, heavy load can cause important host congestion. If too many tasks are assigned to a host, and many of the tasks are waiting for resource allocations, the system resources can become very busy, and the host becomes very slow and finally tasks can not be executed on that host. Table 4.14 presents the failure numbers and the total test numbers for the five algorithms.

Table 4.14 Ratio of success in heavy load state

Algorithm	Failure Number	Total Number	Ratio of Success
Random	9	35	74.29
Shortest	4	31	87.09
Shortest-Capacity	2	19	89.47
Shortest-CPU	4	21	80.95
Threshold	6	25	76.00

4.9 Result analysis on the heterogeneous environment

Our project aims to be used in the industry field with large scientific computations, long tasks play an important role in most applications where plenty of host resources are used in the executions of the legacy programs. Network delays are small, and

occupy a very little part when compared to the utilization of host resources for task processing. We feel that in this context, representing the network cost in addition to the hosts cost would not affect the system performance significantly. In order to simplify the system model, we ignore the network delays in our decisions for system load balancing.

Our goal is to analyze and compare the performance between static load balancing algorithms and dynamic load balancing algorithms. The Random method is the representative of the static algorithm. The other four algorithms are the dynamic load balancing algorithms, but they use different load indexes for the host load estimation. The following discussion can be related to similar, yet more restricted, discussions by Wang and Morris, 1985 who implemented the Random and Join the Shortest Queue algorithms and analyzed their performance in different system loads, and a paper by Rommel (Rommel, 1991) who analyzed three algorithms (Random, Threshold, Shortest) in a homogeneous distributed system.

4.9.1 Average response time

Tables 4.8-4.13 show the system performance of different algorithms under various system loads.

4.9.1.1 Influence of workload on system performance

Looking at figure 4.3, when the system is in a heavily loaded state, the performance of all five algorithms becomes poor. When every host is very heavily loaded, no tasks can be processed normally. All tasks are blocked in the system, and the response time increases significantly. In this situation, the Threshold algorithm

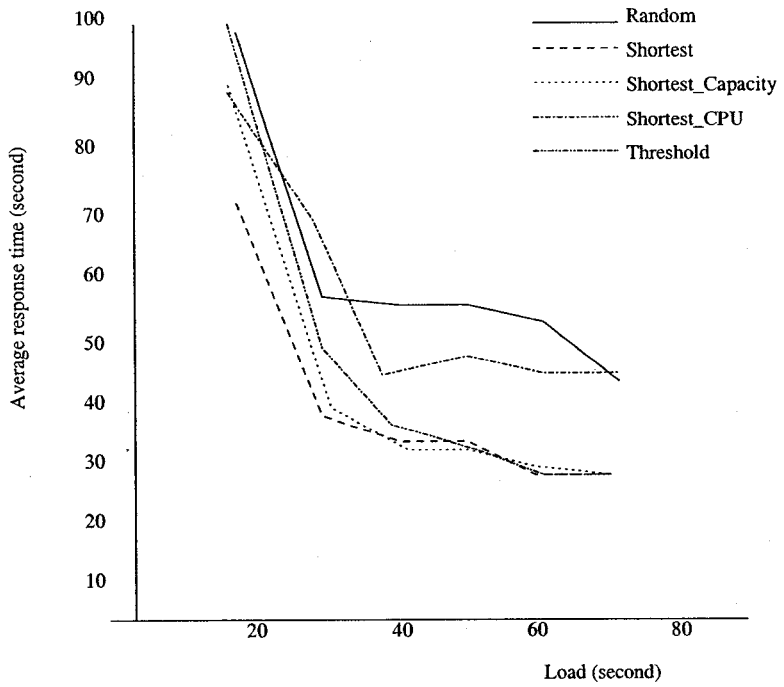


Figure 4.3 Average Response Time.

switches to the Random method. This is the reason why the Random and Threshold algorithms perform the same when system is heavily loaded. Because the Shortest-CPU method does not know any information about the memory and I/O resources, the assignment of the memory-bound and I/O-bound tasks are performed randomly. For this reason, the Shortest-CPU method performs badly when system is heavily loaded.

The system load information is updated periodically. When the frequency of task entering becomes large, the load information can not reflect the real situation when task assignments occur. Based on this old load information, mislead task assignments can be observed. Comparing the Shortest-capacity method with the Shortest method, too much tasks are sent to powerful hosts, and thus degrade the system performance. This cause the Shortest-capacity method to show a worse performance than the Shortest method. The Shortest also yields poor performance

when the system is heavily loaded. But even as it tries to dispatch the tasks load evenly in the system, it still can get performance improvement over the other four algorithms.

4.9.1.2 Load indexes influence on response time

Looking at figure 4.3, and considering the average response time of a group of tasks, dynamic load balancing algorithms all perform better than the random algorithm. The Random method tries to assign system load evenly across the network, but it does not consider the application characteristics and the immediate system status. Because the system is heterogeneous, it may become unbalanced after a period of execution. The Random algorithm can not recognize this situation, and still assigns tasks randomly. It is very easy to achieve a situation where some hosts are heavily loaded while others are lightly loaded. Many tasks with different resource requirements are running on the system, they need different processing time, different memory and I/O resources. The Random method can not use the system resources according to their utilization status. Wang and Morris, 1985 have discussed the performance of the Random algorithm. The main reason for its poor performance lies in the fact that the Random method has no knowledge of the system state information. It can not adapt to fluctuations in system load.

Comparing the Shortest-CPU algorithm with the Shortest, Threshold and Shortest-Capacity algorithms, we can see that the Shortest-CPU method is worse than the others. The Shortest-CPU algorithm only uses the CPU load information as the load index while the other three algorithms use all the CPU, memory and I/O load information as their load indexes. The Shortest-CPU algorithm can thus not assign hosts effectively when memory-bound and I/O-bound tasks arrive. Some CPU lightly loaded hosts may have a heavily loaded I/O system. When I/O-bound tasks

are sent to these hosts according to the CPU index, the I/O resource becomes a bottleneck of the system and degrades the system performance. The CPU index also has similar problems for memory-bound tasks. Although the Shortest-CPU method can not schedule memory and I/O -bound tasks effectively, it improves the system performance when compared with the Random method. The CPU load can partially reflect the system load and improve the system performance. From this comparison, we conclude that the CPU, memory and I/O resource should be considered when the CPU-bound, memory-bound and I/O-bound tasks are dispatched together.

From figure 4.3, we can also observe that system performance obtained using the Shortest algorithm and the Threshold algorithm are not significantly different from one another when system load is not heavy. Both algorithms use the same load information to aid assignment decision, but different host selection methods. The Shortest method assigns one task to a host with the least load. However the Threshold would check a group of hosts and find a proper one, for which the host load state is below a threshold. That is to say, the host is lightly loaded. The difference between the two methods lies in the fact that the Shortest algorithm chooses the lightest host, and the second method selects the first lightly loaded host it meets among a group of hosts. The Shortest method tries to assign the system load across the network evenly, and the Threshold algorithm tries to make the system in a relatively balanced state. Although the threshold method can not reach an absolutely balanced state, it prevents the entire system from becoming heavily unbalanced and keeps the system load in a controlled distribution. Trying to put a task on the most lightly loaded host or just on a lightly loaded host does not make a big difference on the overall system performance. This phenomenon has been mentioned by Eager *et al.*, 1986. In that paper, this result is obtained using a simple analytic model and simulation results in a homogeneous distributed

system; in his conclusions he suggests that "extremely simple load sharing policies using small amounts of information perform quite well" and "nearly as well as more complex policies that utilize more information"(Eager *et al.*, 1986). This analysis is in some sense confirmed by the current experimental results.

4.9.1.3 System capacity influence on response time

From figure 4.3, no obvious performance differences can be observed among the Shortest, Threshold and Shortest-Capacity algorithms when system is not heavily loaded. All these methods use the same current system load information. The Shortest and Threshold algorithms do not account for differences in processing power (CPU speed, Memory size, I/O capacity) among hosts, while the Shortest-Capacity algorithm utilizes the host's capacities to aid in task assignments. From this figure we could conclude that both algorithms yield a better performance than the Random algorithm, but that the use of host capacities can not significantly improve the overall system performance. The current host load information (CPU Idle, Free Memory, I/O utilization) are sufficient to represent the system load status.

It is worth noting, though, that the Shortest-Capacity algorithm is the one that exhibits the highest success ratio among the five algorithms (see Table 4.14). While it does not improve overall system performance, incorporating system capacity into the decision process slightly improves system stability under heavy load.

4.9.2 Mean slowdown

Figure 4.4 presents the mean slowdown of a group of tasks for the five algorithms in our heterogeneous system.

4.9.2.1 Load indexes influence on mean slowdown

Comparing the slowdown of Shortest-CPU and Random algorithms, the Shortest-CPU method is better than Random when system is not heavily loaded. We can conclude that using the CPU load state can enhance performance over no load information being used. The Shortest-CPU and Random algorithms can get good slowdown performance when the system experiences heavy load, as showed in figure 4.4. But the two methods are not stable when working under heavy load. During our experiments, we have found that these two algorithms are the two most likely to yield a congested system(§ 4.9.4).

From figure 4.4, the Shortest algorithm exhibits more stable than the Random and Shortest-CPU algorithms. The Shortest method uses the CPU, memory and I/O load status when tasks are assigned. This method is more apt for finding lightly loaded resources than the Random and Shortest-CPU (where only CPU load is used) methods. With various kind of tasks (CPU-bound, memory-bound and I/O-bound), using all the resource load status (CPU, memory and I/O) is more effective than only using part of the resource status (only CPU).

4.9.2.2 System capacity influence on mean slowdown

From figure 4.4 we can see that the Shortest-CPU, Random, and Shortest algorithms get better slowdown performances than the Shortest-Capacity and Thresh-

old algorithms.

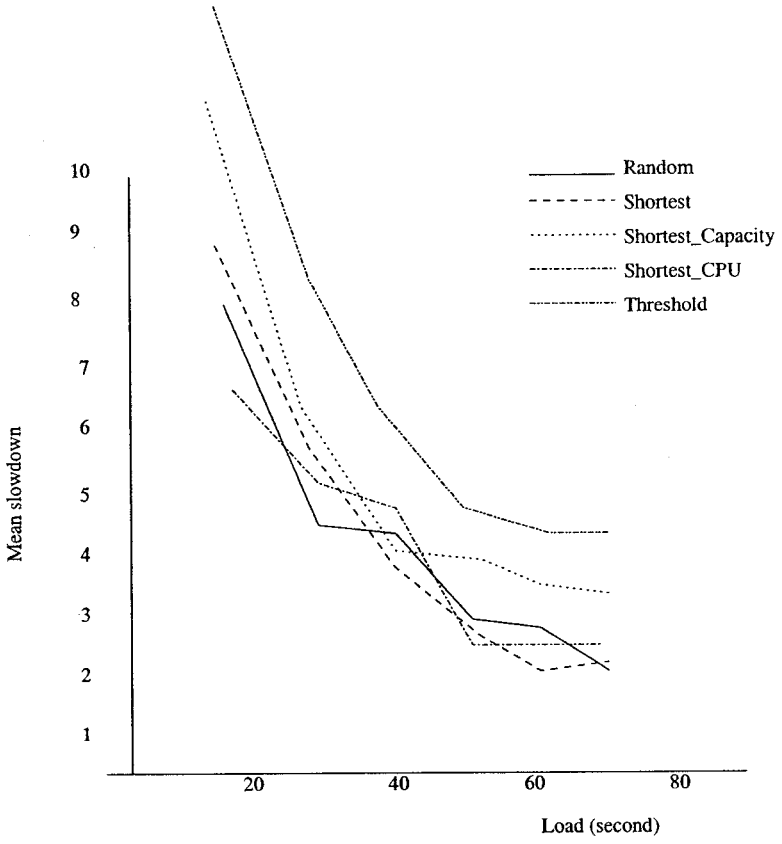


Figure 4.4 Mean Slowdown.

In our experiment environment, the *host5* has more powerful capacities than other hosts. Analyzing our experiment results (for example, the mean cpu used from tables 4.8-4.13), we see that the Shortest-Capacity and Threshold algorithms assign more tasks on this powerful host. This powerful host processes tasks with less CPU time than the other five hosts. As more tasks are assigned to the *host5*, the average slowdown (*elapsed-time / cpu-time*) of tasks becomes longer. However, this host has a strong processing capacity, so tasks get done quickly. The Shortest-capacity and Threshold methods thus have a good average response time (figure 4.3) but a bad slowdown performance (figure 4.4). This could be related to the attitude of a human user, who generally wishes to obtain his result as fast as possible, and

is less interested in the theoretical performance of the system than in the actual response time. The Shortest-Capacity uses the system capacities in the host load evaluations, the more powerful hosts then get selected more frequently. Tracing the Threshold method, the powerful *host5* is listed first in our host group, and it is thus checked first in every assignment. That is the main reason why the two algorithms send more tasks to this powerful host.

4.9.3 Comparing the Average response time with Mean slowdown

The Shortest algorithm reaches good performance in terms of average response time and mean slowdown for a group of hosts. This method not only uses the most powerful hosts but also utilizes the other normal hosts. It is not like the Shortest-Capacity method which readily sends tasks almost exclusively to the more powerful hosts. The Shortest method does not cause delays related to the use of the most powerful host (*host5*). It tries to use each host evenly, and thus gets a good slowdown performance. Because it uses CPU, memory and I/O resource states to evaluate each host, the Shortest method is more inclined to use the proper hosts than the Shortest-CPU and Random algorithms, and gets a better response time performance.

From figures 4.3 and 4.4, we can conclude that using the host status is more effective than no load status while tasks are being assigned. For the workloads where CPU-bound, memory-bound and I/O-bound tasks are mixed together, using all three resource load status (CPU, memory and I/O) is better than only using part of the resources (CPU). Using the system capacities in a load balancing algorithm biases task assignment toward more powerful hosts in a system with high heterogeneity.

4.9.4 Comparison of success ratio

Figure 4.5 presents the ratio of success of all five algorithms in our experiments. In our experiments, when more tasks arrive into the system and can not be processed at once, heavily loaded hosts may become congested. One reason is that the entire system is heavily loaded, and that hosts no longer have enough resources to process incoming tasks. Another reason might be that the load balancing algorithm can not assign tasks evenly in the network and cause some hosts to become overloaded. Because the system load information is updated periodically, the old information may not reflect the real system loads and mislead task assignments. Since the algorithms work in different ways and have different performance, their ratio of success varies for the same system environment. As illustrated in figure 4.5, the Shortest and Shortest-Capacity algorithms have a higher success ratio than the Random and Threshold algorithms. The Shortest-CPU algorithm is in the middle of them.

Because the Shortest-CPU and Random algorithms can not evaluate resource status, the system can be congested due to bad assignments. For the I/O-bound tasks, the Shortest-CPU method uses the CPU load state to schedule them, hosts may be congested for the I/O resource, while CPU load state is not heavy. The Random algorithm does not use the system load information to assign tasks, if the system load is not in a balanced state, the assignments on heavily loaded hosts cause these hosts to become overloaded, which leads to congestion.

The Shortest and Shortest-Capacity algorithms try to find a host which is lightly loaded, and the system better maintains a balanced state. These two methods thus exhibit a better success ratio than the other three methods.

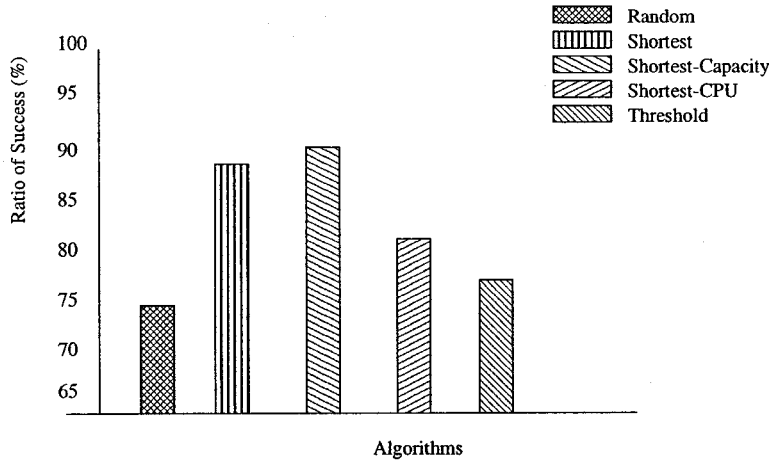


Figure 4.5 Ratio of success.

4.10 Comparison of the homogeneous and heterogeneous environment

The results obtained on the high speed homogeneous system and the heterogeneous system (§ 4.9.1) are very different. The system performance of the five algorithms in a heterogeneous system differ significantly. But the system performance of these algorithms in a high speed homogeneous system differ very little. That is to say that not only the system processing power influences the performance of the load balancing system, but also the distribution of processing power among nodes.

In an heterogeneous system, each host is independent of others and its configuration varies largely. As time passes, the system can easily reach an unbalanced state. The load balancing algorithms try to pull the system to the balanced state based on various kinds of load informations. The effectiveness of the algorithms influences importantly the overall system performance. In this context the Shortest, Shortest-Capacity, Shortest-CPU and Threshold algorithms can improve the entire system performance when compared to the Random algorithm. Because the four algorithms have different abilities to maintain the system balanced, system performance differs largely based on each of them.

In a homogeneous and high power system, tasks do not delay for long at hosts. Each host has a strong processing power, tasks can be processed immediately and the system unbalanced state can be avoided. Sophisticated load balancing algorithm thus become less necessary in such a system where all hosts are in the same or similar states, and for which a simple Random algorithm seems to perform correctly.

However, when more tasks enter the system in a fixed interval of time and when system loads become heavy, the performance of all algorithms in both of the experimental environments degrades obviously. The Random algorithm even shows a better performance than dynamic load balancing algorithms which use the system load informations to help the decision-making process. At this point, the load balancing algorithms are using load informations that is too old and that misleads the task assignments. In order to update system load appropriately, the period of system load collection should be shortened. But the frequent load collection and sending can impede system performance and make the situation worse. The Threshold algorithm could be modified to queue tasks when a heavily loaded state appears. The fatally overloaded state of the system thus could be avoided, and the system would be protected from crashing down because of a situation where too many tasks simultaneously enter the system.

Based on the comparison of the two experimental environments, we can conclude that our load balancing system is most suitable for a system where the load is in middle state. When the system is over loaded, the load balancing algorithms can not assign tasks correctly. As for when the system load is very light, the load balancing algorithms are not necessary.

CONCLUSION

The VADOR project is a MDO application framework integrating different legacy programs to be executed in an unique system environment to help engineers from different departments to cooperate and work together. This system is running under a local network with various kinds of workstations based on different operating systems. Lots of legacy softwares are wrapped in the virtual environment. In order to run these softwares effectively, logical combinations of softwares are implemented in VADOR, such as: sequence, parallel, loop and branch. In order to improve the system utilization, system optimization is imported into this project.

The load balancing subsystem is a part of VADOR which dispatches tasks evenly in the distributed system. The purpose of the subsystem is to prevent situations where some hosts are heavily loaded while others are in light or idle status in the network. The design and implementation include system architecture and load balancing algorithms. Many architectures have been discussed and different algorithms presented. In VADOR, the central system architecture is implemented with several classical placement policies (Random, Shortest, Threshold etc). The performance comparison of different algorithms is outlined in our project.

Architectural design

The load balancing system uses centralized management of load informations and a scheduler to assign tasks. A *global collector* aims to gather host loads across the network and stores them in its internal data structures. All new incoming tasks are sent to the *global collector* and wait assignment there. The *global collector* uses the system load informations and evaluates each host to make a choice based on

the placement algorithm. *Local collectors* are installed on every host of the system. They collect the load information of the local host periodically and send them to the central scheduler (global collector). Once the *local collector* picks up the host load, it compares it with the previous values. If the host load changes significantly from the previous collection, the new data is sent to the global collector, otherwise the collected data is discarded. The advantages of the periodical collection in local collectors include:

- Only the necessary information is sent to the global collector;
- It reduces the communication cost between local collectors and the global collector;
- The global collector saves the host evaluation time for task assignments;
- It allows to easily configure load collection according to workload characteristics.

This architecture is simple and easy to understand. Load balancing algorithms (placement policies) are implemented in the global collector. All tasks are scheduled centrally and the load balancing algorithm is called to choose a host when the assignment happens. The implementation of load balancing algorithms is independent of the system load collection. When the system wants to change the load balancing algorithm, a message is sent to the global collector and the other part of the load balancing system is not influenced. This is a convenient way to manage the load balancing algorithms and monitor the system loads. All system load informations are gathered in the global collector, the system monitor only needs to communicate with the global collector to read the system loads and display them to the user. Communication costs are saved by not referencing all hosts across the network.

A three level application model is incorporated in the architectural design.

System Monitor <—> Global Collector <—> Local Collector

The protocols and services are defined between *SystemMonitor* and *GlobalCollector*, *GlobalCollector* and *LocalCollector*. This model makes the load balancing system easy to develop and maintain.

System implementation

The VADOR project is used as the framework for the implementation of a heterogeneous distributed system. Java was chosen as the programming language because of its cross-platform capability, programming flexibility and strong communication features. Plenty of Java tool packages are supplied freely and used widely in many industry fields, especially for distributed system design.

In the load balancing system, local collectors are installed on each host and collect host load informations. System commands are used for information collection. The implementation of system commands distinguishes between different operating systems, even though all of them are flavors of Unix. They usually have different input parameters and various output formats. In our project, these commands are wrapped in a unique interface. Our network system is based on different Unix workstations running a number of operating systems, among which IRIX and Linux where mainly used for test purposes. *Shell* scripts where used to encapsulate system commands and to format their outputs. The wrapped commands include the host load collection commands and program execution information collection commands.

The local collector picks up host load periodically. The parameters (such as: collection frequency, interval time, and sample count) are initialized and set in the

shell batch files and used by the wrapped system commands. Users can modify these parameter values manually and restart the local collector to activate the new settings. Because of the independence between local collectors, the parameter settings of a local collector has no influence on others. This gives users more flexibility to configure each local collector.

The encapsulation of system commands presents the following advantages:

- Easy to modify, update and maintain system functions;
- Easy to configure load balancing system parameters;
- Easy to extend to a new system environment;
- Programs independent from the operating systems, improve the portability of codes.

Algorithms comparison

Load balancing algorithms can utilize the system resources efficiently and prevent waste of available system resources. Various algorithms for host selection have been discussed in previous research. The load balancing system developed in the VADOR project aimed to improve system performance. Five classical algorithms were implemented in our system: Random, Shortest, Shortest-Capacity, Shortest-CPU and Threshold. In the load balancing system, not only were system resources considered, but also application characteristics used in decision-making process. When a task arrives, the attributes of the task are estimated (CPU-bound, memory-bound, or I/O-bound), this information is used to decide which system resource (CPU, Memory, I/O) should be used in the load evaluation. In a computer system, memory has an important influence on the system performance. If the system

does not have enough memory, frequent page ins and outs degrade the system performance. For a incoming task, memory requirements are first used to select hosts.

In our system the dynamic load balancing algorithms (Shortest, Shortest-Capacity, Shortest-CPU and Threshold) have a good performance improvement when compared to the Random method with the average response time. This demonstrates that the utilization of system load status helps to use the system potential resources and improve the system effectiveness. The Shortest and Threshold methods do not show large differences with respect to system performance. This result is compatible with the phenomenon discussed in the paper by Eager *et al.*, 1986.

Comparison of dynamic load balancing algorithms between methods that consider host system configuration capacities and methods that do not consider the system capacities, show almost no difference in performance measured by average response time. This shows that current system load status is generally sufficient to determine host selections. For two hosts with similar load status but different capacities, when both of them are in light load, each task can execute quickly, the advantage in the high capacity host is not very obvious. On the other hand, if both of them are heavily loaded, the high capacity host processes tasks more quickly, and both algorithms can detect the state change in the powerful host and send more tasks to it. The difference between the two methods is that more tasks are able to be scheduled on the powerful host in the method which considers the system capacities. That is to say, the potential capabilities of the powerful host can be used with a higher priority.

For the dynamic load balancing algorithms with different load indexes, using more resource load status (CPU, memory and I/O) can be more effective than only using part of the resource load status (CPU) for our workloads which mix the

CPU-bound, memory-bound and I/O-bound tasks together. CPU load only can not reflect resource load status for memory-bound and I/O-bound tasks. The algorithm which only uses the CPU load can not assign tasks effectively in the case of memory-bound and I/O-bound tasks.

From the comparison of algorithms between an heterogeneous system and an homogeneous system with powerful workstations, we conclude that the dynamic load balancing algorithms exhibit better system performance than the Random algorithm for the heterogeneous system. In the homogeneous system with powerful hosts, the dynamic load balancing algorithms and the Random algorithm have almost the same behavior. If the system has a very powerful processing capacity, tasks do not stay for long on each host, and the system can be maintained in a balanced state. In that case, the dynamic load balancing algorithms have little influence on system performance. In either of the two system environments, if tasks enter the system more frequently, at some point the system becomes heavily loaded, and the load balancing algorithms can no longer maintain good system performance. In such an extreme case, the Random algorithm may even achieve a better system performance than some dynamic load balancing algorithms.

Future

Our project aimed to implement a load balancing subsystem for the VADOR project and develop several algorithms to place tasks evenly across the network. The Random, Shortest, Shortest-Capacity, Shortest-CPU and Threshold algorithms are coded in our system, and performance comparisons have been made between them for various workloads.

We have tested all algorithms several times using semi industrial workloads. The

workloads are organized based on our own working experiences, and further tests should be done in a real working environment in Bombardier. System execution would be traced and the running data would be logged for a long period to analyze the algorithms performance in the work place.

For the Threshold algorithms, our current implementation puts the powerful host to be evaluated first, which lets the powerful host be used more often. The influence of host order on the overall performance of the algorithm should be investigated.

Our load balancing system does not currently queue tasks if the system is overloaded. Among our five algorithms, only the Threshold algorithm can be used to queue tasks. By testing the entire system, the Threshold algorithm can know that all hosts are heavily loaded and then queue the incoming tasks. Based on new information this algorithm could test the system load again. This could prevent system congestion in heavy load situations.

A new Threshold-Capacity algorithm can be investigated to use system power effectively. In this algorithm, for a group of lightly loaded hosts (host load below the threshold value), the host with the most powerful capacity is selected. This algorithm tries to use the powerful host more frequently without user interventions.

Local collectors gather system load information periodically. Three parameters are used in our system: collection period, sampling count, and sampling interval. The influence of these parameters needs to be considered in the system performance. Intuitively, the sampling interval and count determine the precision of the evaluation. If the sampling interval or count increases, more history information is collected, otherwise only the most recent information is available. If more history information is collected, it may not reflect the current real host load. However, if too little history information is collected, the value is easily influenced by some

sudden events and the real host state can not be well represented by the value. The relationship between the sampling period and the system performance should be investigated through further experiments.

The characteristics of applications influence system performance. We have prepared CPU-bound, memory-bound and I/O-bound workloads to run in our system to see the system responses for all load balancing algorithms. In our load balancing algorithms, the CPU load is used for CPU-bound tasks, the free memory is used for memory-bound tasks, and the I/O load is dealt with for the I/O-bound tasks. From the experimental results, the load balancing algorithms which use more resource load information (Shortest, Shortest-Capacity and Threshold) work not only better than the Random method, but also have obvious performance improvement when compared with the single element load index (only CPU load in Shortest-CPU method). In our current workloads, the CPU-bound tasks represent a large percentage of the global application set. Different workloads where the three kinds of tasks play various roles should be tested for using our algorithms and compared for performance.

REFERENCES

ALZUBBI, A., NDIAYE, A., MAHDAVI, B., GUIBAULT, F., OZELL, B. and TRÉPANIER, J. (2000). On the use of JAVA and rmi in the development of a computer framework for MDO. *8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization.*

BERMAN, F., WOLSKI, R., FIGUEIRA, S., SCHOPF, J. and SHAO, G. (1996). Application-level scheduling on distributed heterogeneous networks. *Supercomputing '96.*

CALZAROSSA, M. and SERAZZI, G. (1993). Workload characterization: A survey. *Proc. IEEE, 81*, 1136–1150.

CAO, J., BENNETT, G. and ZHANG, K. (2000). Direct execution simulation of load balancing algorithms with real workload distribution. *The Journal of Systems and Software*, 227–237.

CHEN, P. M. and PATTERSON, D. A. (1993). A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performence. *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* Santa Clara, CA, USA, 1–12.

CHEN, P. M. and PATTERSON, D. A. (1994). Unix i/o performance in workstations and mainframes. *University of Michigan CSE-TR-200-94.*

CROVELLA, M. E. and HARCHOL-BALTER, M. (1998). Task assignment in a distributed system: Improving performance by unbalancing load. *Sigmetrics '98 Poster Session.*

DEVARAKONDA, M. V. and IYER, R. K. (1989). Predictability of process resource usage: A measurement-based study on unix. *IEEE Trans. on Software Eng.*, 1579–1586, vol 15, no 12.

DIMARCO, J. (1997). Spec list. *University of Toronto, http://www.tangent.org/ brian/dict/hpwww.epfl.ch/bench/SPEC.html.*

EAGER, D. L., LAZOWSKA, E. D. and ZAHORJAN, J. (1986). Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, 662–675, vol SE–12, no 5.

EZZAT, A. K. (1986). Load balancing in nest: A network of workstations. *Proceedings Fall Joint Computer Conference, Dallas.*

FERRARI, D. and ZHOU, S. (1987). An empirical investigation of load indices for load balancing applications. *Performance '87, the 12th Int'l Symp. on computer Performance Modeling, Measurement, and Evaluation*, 515–528.

GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

GOSWAMI, K., IYER, R. and DEVARAKONDA, M. (1989). Load sharing based on task resource prediction. *Proc. 22nd Annu. Hawaii Int. Conf. System Sciences,Kona, Hi.*

HARCHOL-BALTER, M. (1999). Task assignment with unknow duration. *Technical Report in School of Computer Science, Carnegie Mellon University.*

HARCHOL-BALTER, M., CROVELLA, M. and MURTA, C. (1999). On choosing a task assignment policy for a distributed server system. *IEEE Journal of Parallel and distributed Computing*, 59:204–228.

HUI, C.-C., CHANSON, S. T., CHUI, P.-M. and LAU, K.-M. (1995). Balance – a flexible parallel load balancing system for heterogeneous computing systems and networks. *Technical Report HKUST-CS95-42.*

JOHNSON, K., COZZETTO, C., ZURSCHMEIDE, J. B. and RAITHEL, J. (1999). *IRIX Admin: System Configuration and Operation.* Silicon Graphics, Inc.

KOTSIS, G. (1997). A systematic approach for workload modeling for parallel processing systems. *Parallel Comput. 22*, 1771–1787.

KUNZ, T. (1991). The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering, 17*, 725–730.

LU, C. and LIU, J. C. (1996). Distributed scheduling framework- a load distribution facility on mach. *International conference on Parallel and Distributed Processing Technique and Application.*

MCCALPIN, J. D. (1995). Sustainable memory bandwidth in current high performance computers. *Advanced Systems Division, Silicon Graphics, Inc.*

MCVOY, L. W. and STAELIN, C. (1996). lmbench: Portable tools for performance analysis. *USENIX Annual Technical Conference.* 279–294.

MEHRA, P. and WAH, B. W. (1993). Automated learning of workload measures for load balancing on a distributed system. *ICPP*, 263–270.

MemmanageHP (2003). Managing memory performance. http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V50_HTML/ARH9GATE/CHVMXXXX.HTM.

Memsize (2003). Memory size and system performance. http://www.pcguide.com/ref/ram/size.htm.

Mipscpu (2003). Mips/mflops and cpu performance. http://futuretech.mirror.vuurwerk.net.

MITZENMACHER, M. (1997). How useful is old information? *ACM Press New York, NY, USA*, 83–91.

MUSLINER, D. J. and BODDY, M. S. (1996). Contract-based distributed scheduling for distributed processing. *AAAI.*

Netlib (2002). Netlib repository at utk and ornl. http://www.netlib.org.

PASQUALE, B. K. and POLYZOS, G. C. (1993). A static analysis of characteristics of scientific applications in a production workload. *Processing of the 1993 conference on Supercomputing*, 388–397.

PASQUALE, J., BITTEL, B. and KRAIMAN, D. (1991). A static and dynamic workload characterization study of the san diego supercomputer center cray x-mp. *In Proc. ACM SIGMETRICS Conference*, 218–219.

RABENSEIFNER, R. and KONIGES, A. E. (2001). The parallel communication and i/o bandwidth benchmarks: b_eff and b_eff_io. *proceedings of the CUG SMMIT 2001, Indian Wells, California, USA.*

ROMMEL, C. G. (1991). The probability of load balancing success in a homogeneous network. *IEEE Trans. on Software Eng.*, vol 17, 922–933.

SCHROEDER, B. and HARCHOL-BALTER, M. (2000). Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. *9th IEEE Symposium on High Performance Distributed Computing.*

SELLERS, D. (2001). Amd wants to replace mhz with tpi measurement. http://maccentral.macworld.com.

SHARP, O. and BACON, D. F. (2001). The spec cpu benchmarks provide a standard yardstick for comparing performance across platforms. http://www.byte.com.

SHIRAZE, B. A. and HURSON, A. R. (1995). *Scheduling and Load Balancing in Parallel and Distributed Systems.* ISBN 0-8186-6587-4. IEEE Computer Society Press.

SVENSSON, A. (1990). Load sharing in distributed system. *Proceedings of 10 International Conference on Distributed Computing Systems*, 546–553.

WANG, Y.-T. and MORRIS, R. J. (1985). Load sharing in distributed system. *IEEE Transactions on Software Engineering, c-34*, 204–217.

WASSON, S. (2002). An update on amd's true performance initiative. http://www.tech-report.com.

XIAO, L., CHEN, S. and ZHANG, X. (2002). Dynamic cluster resource allocations for jobs with known and unknown memory demands. *IEEE Transactions on Parallel and Distributed Systems, 13*, 223–240.

XIAO, L., ZHANG, X. and QU, Y. (2000). Effective load sharing on heterogeneous networks of workstations. *Proceedings of 2000 international Parallel and Distributed Processing Symposium.*

ZHOU, S. (1987). An experiment assessment of resource queue lengths as load indices. *Univ. California, Berkeley, CSD-86-298.*

APPENDIX I

TASK EXECUTIONS

VADOR tasks are created through the VADOR GUI illustrated in Figure I.1, and sent to the *Executive Server* for node assignment. The *Executive Server* invokes the *Global Collector* to select a host using a load balancing algorithm. Tasks are then assigned to the selected host to run.

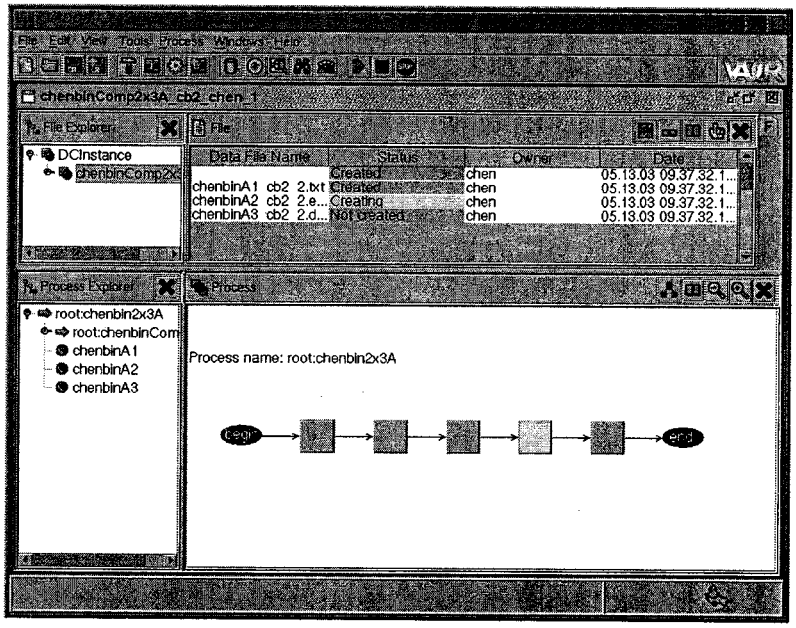


Figure I.1 VADOR GUI.

I.1 Task creation

On the VADOR GUI, the *new* button is used to create new tasks. These new created tasks can be saved in the database by clicking the *save* button. Users can open an existing task with the *open* button and run this task in the future.

I.2 Task execution in interactive mode

Through the VADOR GUI, tasks can be assigned manually. When a task is opened through the *open* button, it appears in the GUI window. Users can execute this task by clicking the *execute* button. When the *execute* button is selected, the chosen task is sent to the *Executive Server* where the *Global Collector* is invoked and a load balancing algorithm is used to choose a host to run the task. The execution information will be sent back to the *Executive Server* and saved into the database. When the task execution finishes, an *Execution over* message is sent back to the VADOR GUI and shown to the user.

I.3 Task execution in batch mode

Through the *Loadbalance Manager*, users can select a group of existing tasks which are created through the VADOR GUI. The task selection interface is shown in Figure I.2.

After a group of tasks are selected, users can use the *schedule* interface to arrange the assignment order and starting execution time of these tasks. Figure I.3 presents the schedule GUI. The interval period between two tasks is set in the column *Delay*. The task's execution order can be moved with the *up* and *down* buttons. Once these tasks are set down, they are ready to run in the VADOR system and can also be saved into the database for future use.

When the *Execute* button is selected, this batch of tasks are sent to the *Executive Server* to run consecutively.

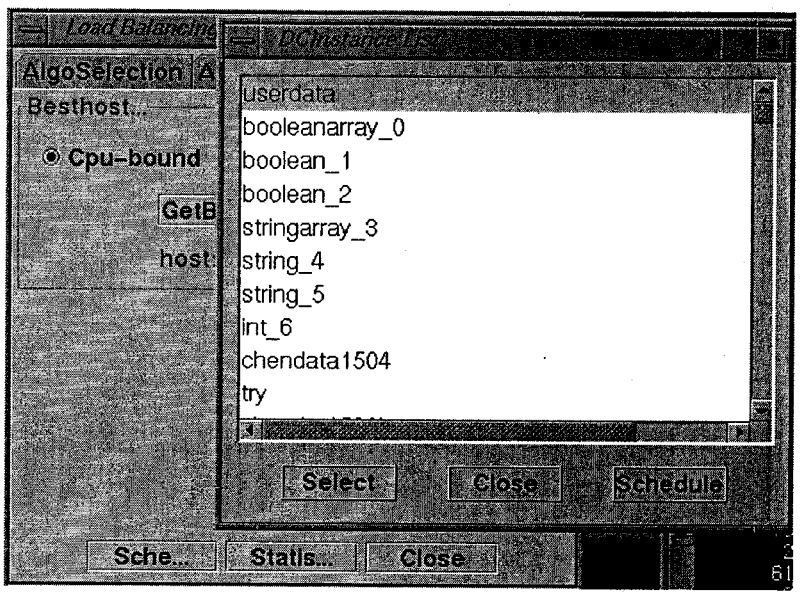


Figure I.2 Task Selection.

I.4 Global Collector configuration

Using the interface illustrated in Figure I.4, the *Global Collector* can be configured to use different load balancing algorithms.

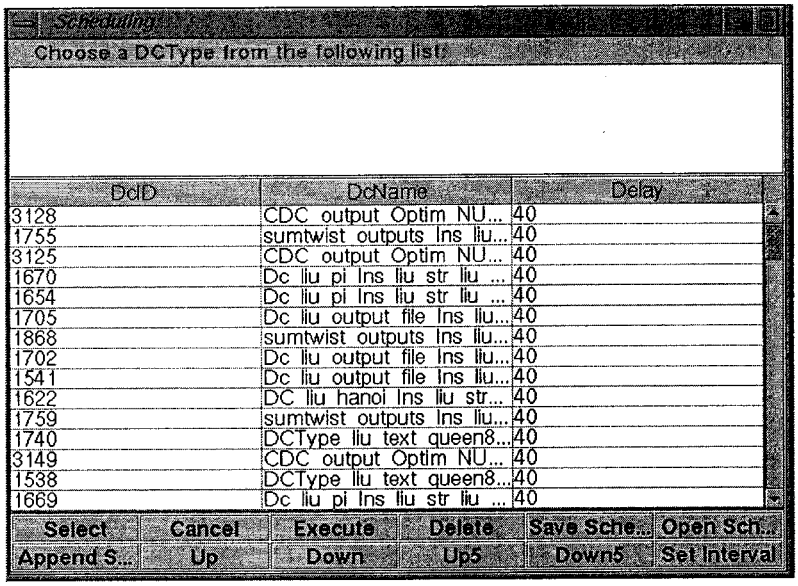


Figure I.3 Task Schedule.

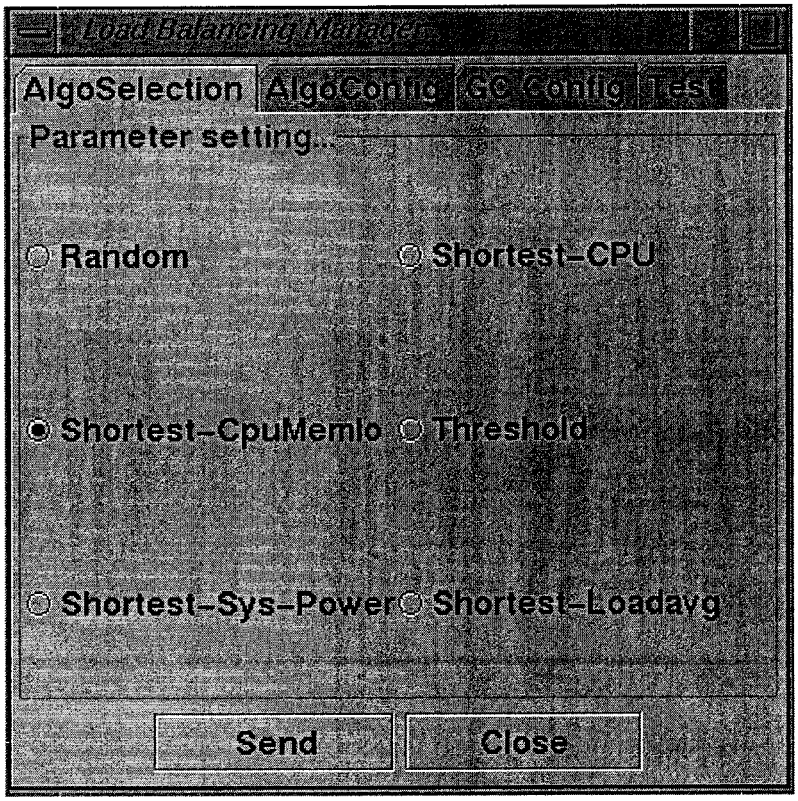


Figure I.4 Algorithm selection.